Strategic Research Roadmap for European Web Security
FP7-ICT-2011.1.4, Project No. 318097
http://www.strews.eu/

Deliverable D1.2

Case study 1 Report: WebRTC

Abstract

Built-in handling of Real Time Media (audio, video) on the web promises potentially significant change in telephony and in conference calling. The W3C WebRTC and IETF rtcweb working groups are developing the set of specifications that will allow browsers and web sites to support such calling and other functions. This is clearly a potentially security sensitive extension to the web, so STREWS has devoted effort on this topic as a case study to both attempt to improve the overall security of the result and to see if this approach holds promise as a way to improve interactions between researchers and standards makers and hence the overall security of the web. In this deliverable, we show some possibly new issues with WebRTC security discovered by researchers (from SAP) that the standards makers may not have considered. However, while this deliverable is, as a deliverable, final, the work itself goes on, partly involving discussions between the STREWS project and participants in the IETF and W3C so in technical terms this remains a work-in-progress. (We have therefore updated this deliverable to take account of developments since the v1.0 version.)

Deliverable details

Deliverable version: *v1.1*
Classification: *public*
Date of delivery: *29.09.2014*
Due on: *M15*
Editors: *Stephen Farrell*
Total pages: *63*

List of Contributors:
Bert Bos, Elwyn Davies, Lieven Desmet, Stephen Farrell, Martin Johns, Rigo Wenning

Partners: ERCIM/W3C, SAP, TCD, KU Leuven

Document revision history

	Responsible	Date
First complete	TCD	February 26, 2014
Outside project version	TCD	April 20, 2014
Initial delivered version	W3C	April 30
Summer 2014 update	TCD	August 30
Camera ready version	TCD	September 21
Internal-review (ED) changes	All	September 25
Final top and tail	TCD	September 29

Executive Summary

WebRTC or Web Real Time Communications allows the establishment of real time audio and video communications between users' browsers. The creation of the communication link is mediated by a web server with WebRTC capabilities. The link itself can be peer-to-peer between the browsers. WebRTC does not depend on a standardised signalling infrastructure, but rather leverages the web JavaScript environment and standardised browser APIs. This allows for implementations that range from a simple audio communication between two people up to a video-conference with multiple participants, out of the box as part of a normal browser. WebRTC thus has at least three actors involved: a web server and two browsers. The mediation of the usable communication channels is negotiated between these based on a complex set of specifications that are detailed in the WebRTC overview below. Not only is there a complex combination of specifications, but the work is also distributed between the IETF doing the protocol stack and W3C creating the browser APIs. Still WebRTC remains a part of the Web, thus all the vulnerabilities described in the Web Security Guide [25] still apply. But the complexity and the fact that the communication is real time also bring new aspects and vulnerabilities.

This case study dives deeper into the vulnerabilities to which user and server assets are exposed by their use of WebRTC. This is not limited to the three actors mentioned, but these may also behave in unexpected ways. Chapter 2 describes the relevant assets, and based on the list of assets, the surface of exposure and the assets targeted and the list of possible malicious actors becomes much clearer. This exposes some of the underlying issues otherwise hidden by the complexity of the combination of WebRTC specifications.

The fact that real time communication is involved immediately raises the issue around permissions and timing of access. For communication between two users, the browser needs access to speaker **and** microphone. This means the browser can potentially be remotely turned into a device to capture all the sounds in a location. It is known that native applications on mobile phones have issues around permissions and how fine grained they ought to be, whereas how coarse they actually are. For WebRTC the issue is the same. This case study assesses the implications and gives hints on potential security problems in current implementations.

A central issue in WebRTC that was discussed widely in the Working Groups but has not yielded satisfactory results is identity management. Am I really sure I am talking to the right person? On the Web, the trusted telephone operator is not always there. The case study looks into user authentication and naming and raises issues around credentials and keying. The case study then goes on with an analysis of a classic cross site scripting attack in a WebRTC scenario. What happens when an attacker can inject arbitrary JavaScript code into the running WebRTC application or even manages to upload a malicious WebRTC application to a server that delivers it to the browser? The evaluation of exposure is used to describe a landscape of possible attacks.

Finally, browser real time communication is just the next generation of telephone communication. As indicated in the introduction, we can call on the values set by the EU Charter of fundamental rights as well as the European Convention of Human Rights (ECHR) to justify a focus on confidentiality leaks. The rules indicate the high level of protection of confidentiality required for real time communication. In chapter 6, the report details where this value may be exposed to unintended consequences.

Contents

List of Figures

Introduction

This document presents the results of the first STREWS case study. For this study, the project decided to investigate Web Real Time Communication services, in the following referred to as WebRTC. WebRTC is a way to enable built-in audio and video communications between users' web browsers, with direct communication of media between browsers and without having to depend on a standardised signalling infrastructure, but rather leveraging the web JavaScript development environment and standardised browser APIs. This will allow for implementations that range from a simple audio communication between two people up to a video-conference with multiple participants, providing these functions "out of the box" as part of the basic capabilities of a normal browser.

One promise of WebRTC is that it may revolutionise communications between consumers and service providers, so that web sites can have a simple "call me" button to allow site visitors to have an audio or video call with site sales or support staff. But WebRTC is not only the basis for new innovative tools, it is also architecturally innovative. WebRTC moves away from the classic client/server paradigm and introduces new elements of direct browser to browser communication into the web. While the signalling is still done in the client/server paradigm, the actual media channel is a direct communication between the browsers without (modulo e.g., firewall traversal via "TURN", described below) further need to pass the media content, the audio or video packets, via a central service. This reduces the importance of the central network points, thus leading to better load distribution. From a security point of view it remains to be seen whether this is a good or a bad thing. Because a telco hosted central switching point is a tempting asset to attack, but often highly secured, while the web server and browser endpoints may reside on machines that are less well protected, but less attractive targets.

It is therefore timely to assess the security architecture of WebRTC as standardisation is ongoing. Additional security assessment by researchers at this stage should help to avoid weaknesses in the architecture and in resulting implementations and deployments. Once the architecture is finalised, if a weakness is discovered, it is very hard to change the installed base that was deployed using the initial architecture. But a good security analysis will also look at the development of applications. Like every Internet technology, WebRTC also has its pool of developers who will create tools to make developing WebRTC applications easier. Once those tools are widespread, they will introduce their own weaknesses into applications. A flawed toolkit can thus significantly exacerbate security weaknesses in the same way that a flawed architecture can. Patching over time will be much harder than getting the toolkit right in the first place.

The decision of the STREWS project to address WebRTC was guided by several considerations.

- First there was already a focus on WebRTC during the creation of the STREWS proposal to help address the difficult security issues of real time communications. This had been determined in conversations with the participants of the respective Working Groups in W3C and the IETF. Those challenges remain on the table. They have not yet been addressed by the Working Groups.
- Second, the Consortium realised that the impact of WebRTC on the Web architecture is potentially very high.

- Finally, WebRTC allows the application of the schema developed in the STREWS Security Guide to a new and upcoming technology with known issues and unknown issues. Since WebRTC is a web technology it is thus subject to the web security model already described in Deliverable D.1.1 [25]

Methods followed

This analysis of the WebRTC platform will follow the model established by the Web platform security guide [25] developed by the STREWS project as Deliverable D.1.1. The basic understanding of the Web ecosystem developed in D.1.1 is tested against the reality of an evolving WebRTC and its implementations. The understanding of the WebRTC model and how it is embedded into the overall Web ecosystem will allow for the identification of assets that are susceptible to attacks. These may be assets that reside on the user side, within the communication protocol, in hardware and on the server side of the communication. Once the assets have been identified, the case study will describe the possible attacker capabilities. Scenarios inspired by the use cases given in the RTCweb use cases [40] will serve as a basis for a concrete and tangible analysis of the possible attacks and the resulting attacker benefits and victim damages.

The method above is structured as follows:

- Review current WebRTC specifications (and relevant academic literature) with a particular emphasis on security and according to the schema set out by the STREWS Security Guide.
- Review the security of currently accessible WebRTC implementations.
- Do an analysis of WebRTC security according to the method developed by the STREWS the Web-platform security guide [25].
- Identify areas where STREWS and/or other research inputs may be timely and useful.
- Carry out some detailed analyses and contributions to specific WebRTC security topics.
- Identify any remaining areas for future research and standardisation on WebRTC security.
- Iterate the above sequence as project resources allow since updated results may be of use to researchers or those participating in the standards process.

In addition, the STREWS consortium also proposed and guest edited a special edition of IEEE Internet Security on WebRTC security, planned for November/December 2014 [24]. The guest editor's introduction to that issue and the papers from the special issue can be considered to complement this case study, in that both are pursuing STREWS' overall goal of encouraging better interaction between researchers and standardisation.

Part I

WebRTC Overview

Chapter 1

The basic functioning of WebRTC

We assume that readers of this report have read Part I of STREWS deliverable D.1.1 [25] which describes the overall architecture of the Web Platform and how its elements relate to one another. Deliverable D.1.1 describes the classic Web model that, in summary, is constituted from resource identifiers (URIs), a transfer protocol (HTTP) and a content format (HTML). The evolution from a simple hypertext system into an application development platform as described in deliverable D.1.1 already hints at the evolution of the Web towards greater complexity.

From the beginning Paul Baran[1], one of the founding fathers of the Internet, imagined that the packet switching he invented would be used for direct real time communications like direct phone calls. This turned out to be more difficult than initially thought. Early technology put a high burden on the network infrastructure for rather poor results.

In D.1.1, the Web-Platform Security Guide, the security anlysis procedure starts by determining the assets involved in the technology under consideration. These assets are valuable resources within the entire web application model, spread over clients, networks and servers. Assets are important to an attacker, since obtaining or controlling them helps him achieve his goals, mainly driven by economic incentives [45]. Identifying these assets is therefore crucial for determining the focus of an attacker in the web application model. Once the assets are determined, an attack tree is created that maps the threats to the assets. This allows a matrix of threats and assets to be constructed giving a comprehensive high level overview that can then be explored in more detail.

The WebRTC Internet Draft on use cases and requirements [41] illustrates the WebRTC architecture with the following example:

> Two or more users have loaded a video communication web application into their browsers, provided by the same service provider, and logged into the service it provides. The web service publishes information about user login status by pushing updates to the web application in the browsers. When one online user selects a peer online user, a 1-1 audiovisual communication session between the browsers of the two peers is initiated. The invited user might accept or reject the session.

Figures 1.1 and 1.2 show the sets of actors involved and some of the protocols used. This basic scenario is then extended in various ways. The audio or video communication can be combined with other applications like multi-party online games where multi-party online communications between the players is also required for team cooperation in the play between two or more teams. But the scenario would not be a web scenario if everybody had to have their account with the same provider. If the two Web servers are operated by different entities, some inter-server signalling mechanism needs to be agreed upon, either by standardisation or by other means of agreement. Existing protocols (for example SIP or XMPP) could be used between servers.

[1] `http://en.wikipedia.org/wiki/Paul_Baran`

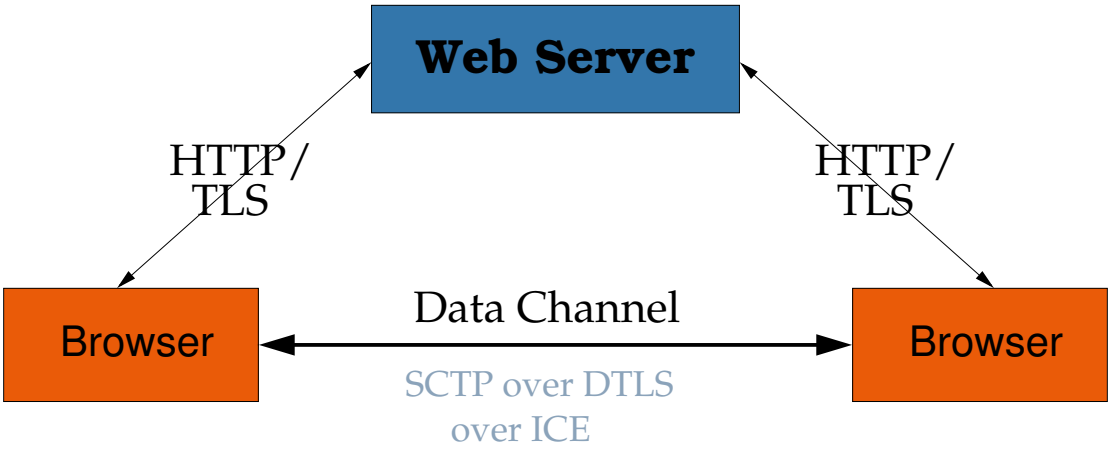

Figure 1.1: Simple WebRTC Architecture at a glance

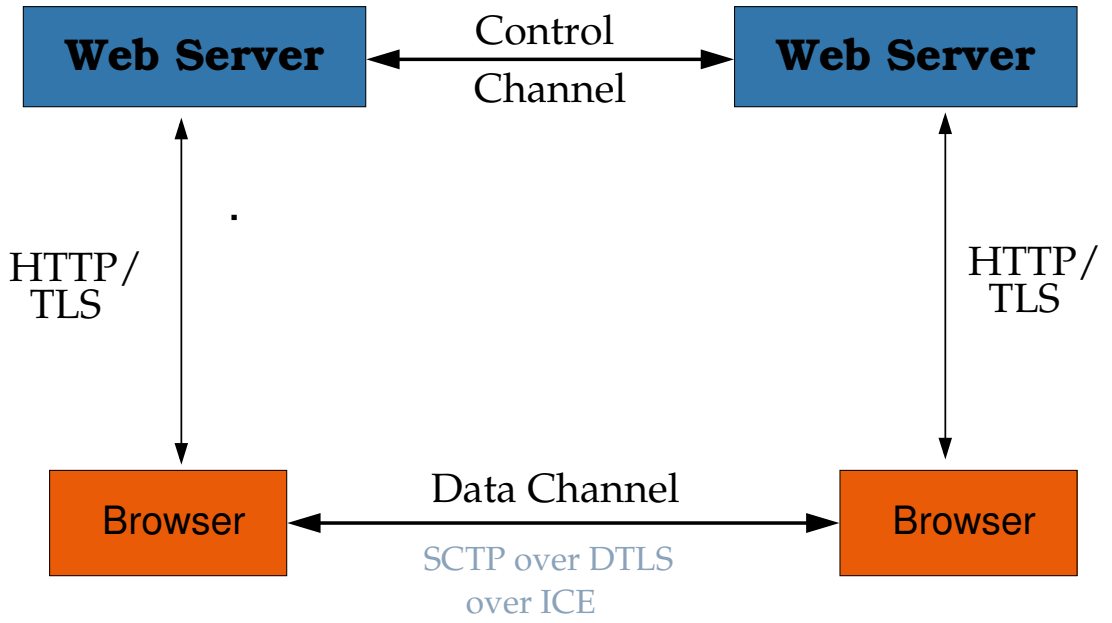

Figure 1.2: Complex WebRTC Architecture at a glance

Each of the browsers exposes standardised JavaScript (JS) calling APIs (implemented as browser built-ins) which are used by the Web server to set up a call between the (user) parties. The Web server also serves as the signalling channel to transport control messages between the browsers. While this system is topologically similar to a conventional SIP-based system (with the Web server acting as the signalling service and browsers acting as softphones), control has moved to the central Web server; the browser simply provides API points that are used by the calling service. As with any Web application, the Web server can move logic between the server and JavaScript in the browser, but regardless of where the code is executing, it is ultimately under control of the server.

WebRTC consists of two distinct standards efforts and components:

1. Protocol specifications developed in the IETF RTCweb working group (WG).
2. JavaScript (ECMAScript) Application Interfaces (API) that control components on the client side, specifications are called WebRTC and are standardised in the W3C WebRTC working group.

While this is a somewhat artificial split, we nonetheless follow it here since the protocol/API split is understood by the WebRTC community and is followed by the extensive W3C documentation and the set of IETF Internet Drafts scheduled to become the protocol specification RFCs in due course.

1.1 WebRTC Protocols

The Web Server delivers content to the browser that allows the browser to initiate the data or communication channel to another browser. The Server controls the communication between the browsers via JavaScript modules by initiating the connection and providing all WebRTC parameters to the browsers. Of course since the call setup is handled via JavaScript, this can include user interaction, but ultimately the web server controls which scripts are executed.

The thinking behind WebRTC call setup has been to fully specify the media plane, but to leave the signalling plane up to the application as much as possible. The rationale is that different applications may prefer to use different protocols, such as the existing SIP (c.f., [49]) or Jingle[2] call signalling protocols, or something custom to the particular application, perhaps for a novel use case. In any case, the crucial information that needs to be exchanged is the multimedia session description [34], which specifies the necessary transport and media configuration information necessary to establish and maintain the media plane.

It is important to note that, except for Network Address Traversal (NAT), conferencing, or similar reasons, the media path goes directly between the browsers, so, in order to achieve interoperability, it has to conform to the specifications of the WebRTC protocol suite. In contrast, the signalling path ("JavaScript over HTTP with TLS") goes via servers that can modify the signals as needed.

According to the WebRTC protocol overview [3] the protocol part requires:

1. **Connection management:** Setting up connections, agreeing on data formats (codecs), changing data formats during the duration of a call.
2. **Data transport:** TCP, UDP and the means to set up connections between entities while meeting the agreed upon security requirements, as well as the functions for deciding when to send data: Congestion management, bandwidth estimation and so on.
3. **Data framing:** RTP (RFC 3550 [73]) and SRTP (RFC 3711 [8]) and other data formats that serve as containers, and their functions for data confidentiality and integrity.
4. **Data formats:** Codec specifications, format specifications and functionality specifications for the data passed between systems. Audio and video codecs, as well as formats for data and document sharing, belong in this category. In order to make use of data formats, a way to describe them, that is a session description, is needed.

Within this architecture, many different paths can be used for the signalling between one or more Web servers and the browsers. While SIP [67] and XMPP/Jingle [69, 70, 68] have been most common protocols for this signalling path to date, customised JavaScript is expected to be the most common in future. However, even if SIP and Jingle are somewhat in the category

[2] `http://xmpp.org/about-xmpp/technology-overview/jingle/`

of "legacy" capabilities all the vulnerabilities of those technologies will also affect the level of security achievable by WebRTC.

The communication channels between a Web server and a browser are well known. They use HTTP URIs via HTTP/1.1 [32] over TCP or HTTPs URIs HTTP/1.1 over TLS (TLS 1.0 [26] or TLS 1.2 [27]). However, with the ongoing development of HTTP/2.0 [9] new combinations will become possible, for example accessing HTTP URIs via HTTP/2.0 over TLS [55]. The content transported is HTML4 [61] or HTML5 [11].

1.1.1 Connection Management

Media session parameters are described using the offer/answer mode of the Session Description Protocol (SDP) [67, 38]. While there is some dis-satisfaction with this, at least amongst some of the web community, no alternatives are under consideration. So at least WebRTC version 1.0 (if versioning is the proper metaphor) is committed to SDP, but future versions may attempt to move away from SDP. The main objection to SDP is that its string representation is hard to manipulate, and it *does* need to be manipulated even in some quite simple use-cases. This adds complexity to the application/signalling layer JS code. While this is a valid criticism, it is not clear that a non-SDP solution would really be easier. However, we ought expect the complexity associated with SDP manipulation to enable some exploits in the wild.

SDP allows servers to convey the necessary information about the communication session endpoints, codecs and some of the security mechanisms required. SDP was originally conceived as a way to describe multicast sessions carried on the Mbone, an experimental IP Multi-Cast network. A complete view of the session requires information from both participants, and agreement on parameters between them. Thus in the general case an SDP description can include many attributes such as:

- Session name and purpose
- Time(s) the session is active
- The media comprising the session
- Information to receive those media (addresses, ports, formats and so on)

As resources necessary to participate in a session may be limited, some additional information may also be desirable:

- Information about the bandwidth required by the session
- Contact information for the person responsible for the session
- The type of media (video, audio, etc)
- The transport protocol (RTP/UDP/IP, H.320, etc)
- The format of the media (H.261 video, MPEG video, etc)

For an IP multicast session, the following are also conveyed:

- Multicast address for media
- Transport Port for media This address and port are the destination address and destination port of the multicast stream, whether being sent, received, or both.
- Remote address for media
- Transport port for contact address

The fact that SDP descriptions can be quite complex and can be bundled together and encompass kinds of session that are not expected to be common in WebRTC (e.g., recordings of TV broadcasts) and that SDP descriptions are new to the web, mean we ought focus on the security aspects of SDP to explore for potential vulnerabilities.

The JavaScript Session Establishment Protocol (JSEP [76]) allows for browser control of SDP description handling. Essentially this interface provides a way to handle the offer/answer model in a way that suits browsers (e.g., considering page reloads) but can also work for other applications. JSEP mainly consists in methods for handling local and remote session descriptions as negotiated by some signalling mechanism, plus a way to interact with the ICE (RFC 5245 [65]) state machine.

1.1.2 Data transport

Another challenge for WebRTC lies in the fact that most browsers are either behind a firewall or within a locally managed network using Network Address Translation (NAT). Internet traffic from a typical internal IP network (using private IP address blocks such as 192.168/16 or 10/8) is funnelled via an external gateway (firewall and/or NAT unit) with a globally routable IPv4 address to the "outside". Frequently such firewalls block most unsolicited UDP traffic from outside and NAT makes the direct addressing of a peer browser more difficult.

To handle such issues, WebRTC uses Interactive Connectivity Establishment (ICE) [65]. ICE is a protocol for NAT traversal for (initially, UDP-based) multimedia sessions established with the offer/answer model. ICE makes use of the Session Traversal Utilities for NAT (STUN) protocol and its extension, Traversal Using Relay NAT (TURN). ICE can be used by any protocol utilizing the offer/answer model, including WebRTC. Consequently, WebRTC requires the implementation of TURN as defined in RFC 5766 [51]. If IPv6 traffic is supported, as will be common, RFC 6156 [17] must be supported as well as STUN, RFC 5389 [66]. All this allows connection either to the servers or between the browser machines and to establish a session.

Since ICE, STUN and TURN are frequently confusing, we re-iterate their functions. STUN is a server operated service that allows nodes behind NAT boxes to find out their public IP addresses and thus assists in establishing connectivity. TURN goes a step further and provides a server through which packets may be exchanged, which enables two nodes that are both behind firewall/NAT boxes to communicate. Both STUN and TURN require a form of authentication between the possibly NATted node and the server. That authentication is based on shared secrets (i.e., currently, passwords), though a new IETF working group ("tram"[3]) has been formed recently to try improve on this. And finally ICE is an algorithm used to establish transport layer connectivity between WebRTC endpoints. ICE essentially tells a node how to create a list of STUN and TURN candidate endpoint addresses and then how to prioritise those so that both sides of a communication arrive at common endpoints. For WebRTC the browser is the canonical ICE endpoint.

For the media channel between browsers, WebRTC implementations must also support SCTP over DTLS-SRTP. A very brief explanation of these protocols may be useful: The Stream Control Transmission Protocol (SCTP) is another transport layer, similar to the much better known TCP or UDP that has been selected for WebRTC. DTLS is the Datagram Transport Layer Security protocol, which securely establishes shared crytographic keys between two hosts so that those keys can be used to secure datagrams. RTP is, of course, the Real Time Protocol, used for transferring media packets. And SRTP is a framework for securing RTP packets, which differs from other security protocols in that with such media, some lost packets can be accomodated without damage to the overall application. DTLS-SRTP as a combination means using DTLS for key establishment (the "handshake") and then using those keys with some SRTP security method.

Considering the protocol stack of Figure 1.1 the usage of DTLS over UDP is specified in

[3] https://tools.ietf.org/wg/tram/charters

RFC 6347 [64], while the usage of SCTP on top of DTLS is specified in Internet-Draft draft-ietf-tsvwg-sctp-dtls-encaps [75]. The Stream Control Transmission Protocol (SCTP) is a transport protocol originally defined to run on top of the network protocols IPv4 or IPv6. In WebRTC SCTP is used on top of the Datagram Transport Layer Security (DTLS) protocol itself running over UDP.

1.1.3 Data framing and baseline security

For transport of media between the browsers, the Real-time Transport Protocol(RTP) [73] is used. RTP itself comprises two parts: the RTP data transfer protocol, and the RTP control protocol (RTCP). In order to support the video conferencing scenario, WebRTC needs support for multiple channels (or sources), thus the need for multiple synchronisation sources (SSRCs).

WebRTC is notable in that it not only requires implementation of security via SRTP (RFC 3711 [8]) and specifically, DTLS-SRTP [53] but also mandates that the secure flavour be the default to offer. This is not a requirement to use the secure flavour, but is quite close to one, and represents an "upgrade" on the usual IETF/W3C level of security requirement, which is to make security features Mandatory-To-Implement (MTI) [71].

Additionally, the SDES [8] method of keying SRTP has not been agreed as being MTI, despite some pressure for that from those most interested in conference calling. The reason to not want SDES is that that exposes the SRTP keying material to the JS code and hence the web server, whereas with DTLS-SRTP the keying material used to protect media is only (easily) available to the media endpoints (i.e., mainly browsers). Nonetheless SDES can still be implemented as an option, and probably will be by some at least. Whether or not browsers will support SDES is not yet clear, so any security analysis needs to consider the SDES option as well.

1.1.4 Data formats

WebRTC wants to allow each communications event to use the data formats that are best suited for that particular instance, where a format is supported by both sides of the connection. However, a minimum standard is greatly helpful in order to ensure that communication can be achieved.

The current development is heavily in flux, mainly because of IPR issues around audio and video codecs.

It seems the IETF Group has achieved some agreement around draft-ietf-rtcweb-audio-05 [78] on the audio codecs that are mandatory to implement. The choice fell on Opus as specified in RFC 6716 [80] and on G.711 PCMA and PCMU with the payload format specified in section 4.5.14 of RFC 3551 [72].

During the second half of 2014, the RTCweb Group were consumed by controversy concerning the video codec, and did not manage to reach rough consensus on which codecs to make MTI. The group is somewhat equally divided over the use of H.264 from MPEG or VP8 as specified in RFC 6386 [6]. Both solutions have their merits and their burden of IPR problems. At the time of writing, the IETF WG have decided to shelve the video codec debate until September 2014 in the hope that market developments will produce a rough consensus for some MTI video codecs.

1.1.5 RTCweb Specifications

Specifications for RTCweb include:

RFCs

1. RFC 2246 The TLS Protocol Version 1.0 [26]
2. RFC 2327 SDP: Session Description Protocol (obsoleted by RFC 4566) [33]
3. RFC 2616 Hypertext Transfer Protocol – HTTP/1.1 [32]
4. RFC 3264 An Offer/Answer Model with the Session Description Protocol (SDP) [67]
5. RFC 3550 RTP: A Transport Protocol for Real-Time Applications [73]
6. RFC 3551 RTP Profile for Audio and Video Conferences with Minimal Control [72]
7. RFC 3711 The Secure Real-time Transport Protocol (SRTP) [8]
8. RFC 4566 SDP: Session Description Protocol (references the obsoleted RF 2327) [34]
9. RFC 5245 Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols [65]
10. RFC 5246 The Transport Layer Security (TLS) Protocol Version 1.2 [27]
11. RFC 5389 Session Traversal Utilities for NAT (STUN) [66]
12. RFC 5766 Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN) [51]
13. RFC 6120 Extensible Messaging and Presence Protocol (XMPP): Core [69]
14. RFC 6121 Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence [70]
15. RFC 6122 Extensible Messaging and Presence Protocol (XMPP): Address Format [68]
16. RFC 6156 Traversal Using Relays around NAT (TURN) Extension for IPv6 [17]
17. RFC 6386 VP8 Data Format and Decoding Guide [6]
18. RFC 6716 Definition of the Opus Audio Codec [80]

Internet Drafts

1. **draft-ietf-rtcweb-use-cases-and-requirements** Web Real-Time Communication Use-cases and Requirements [41]
2. **draft-ietf-rtcweb-audio** WebRTC Audio Codec and Processing Requirements [79]
3. **draft-ietf-rtcweb-data-channel** WebRTC Data Channels [47]
4. **draft-ietf-rtcweb-data-protocol** WebRTC Data Channel Establishment Protocol [46]
5. **draft-ietf-rtcweb-jsep** JavaScript Session Establishment Protocol [77]
6. **draft-ietf-rtcweb-overview** Overview: Real Time Protocols for Browser-based Applications [3]

7. **draft-ietf-rtcweb-rtp-usage** Web Real-Time Communication (WebRTC): Media Transport and Use of RTP [59]
8. **draft-ietf-rtcweb-security** Security Considerations for WebRTC [62]
9. **draft-ietf-rtcweb-security-arch** WebRTC Security Architecture [63]
10. **draft-ietf-rtcweb-stun-consent-freshness** STUN Usage for Consent Freshness [60]
11. **draft-ietf-rtcweb-transports** Transports for RTCWEB [4]

1.2 WebRTC APIs

The complex protocol machinery that was described in Section 1.1 will of course not be exposed to the developer. It serves as a framework for developers to build web applications into web pages using the HTML5 development platform. While the protocols provide the channels, the W3C WebRTC API efforts provide the necessary hooks and APIs to allow developers to use the system for applications that allow for real-time communication.

The real innovative power of WebRTC becomes particularly apparent on mobile phones. The browser in a web application scenario does not have the same look and feel as our desktop browsers. It is mainly a screen with functionality that is controlled by a browser running with little or no browser "chrome" (decoration) around it. One can try that experience by hitting the *full screen* button (typically F11) in a desktop browser. All surrounding window borders and decoration disappears and only the page is rendered. The HTML5 platform means that this apparently pared down environment provides all the capabilities needed for fully fledged application development. Real time communication may thus be only one part of a web application running on a mobile phone. To make the complexity manageable, we will take a step-by-step approach and start with the browser's part in a very basic WebRTC scenario and then increase the complexity and the context.

1.2.1 The Browser interactions

As usual we start with the browser loading a web page that contains relevant information within some JS (or more properly, ECMAScript [43]). The browser can now start a negotiation (via the web server) with another browser in order to establish a Browser-to-Browser connection. To do that it uses the channels given by WebRTC as explained in Section 1.1. While an SDP description doesn't tell the user agent how to exchange the session information, it contains the proposed setup for a session to come. For WebRTC this includes requirements of all kinds like the presence of a microphone, the codecs to be used for the exchange, the STUN/TURN credentials required etc. Now both browsers, identified by the server and controlled via the server signalling channel start to exchange rounds of offers and answers, each of which are processed via callbacks. The JS API defined by the W3C WebRTC specification describes the commands the browser can use to react to those callbacks and where to get the information to put in the response data of the callback dialog.

Once both browsers are in agreement over the transport, framing and content format to use, communication via the media path is opened between the browsers. It is important to bear in mind that the server retains full control over the communication, even if the packets flow from browser to browser directly.

Before media transfer starts, the browsers need to complete the ICE negotiations, and any DTLS-SRTP exchanges. ICE [29] is how WebRTC browsers search for connectivity potentially in the face of firewalls and/or Network Address Translators (NATs) local to each browser. These exchanges might take a number of round-trips, which can be problematic for so-called "early

media," i.e., voice or video packets that are produced before the channel is ready. Handling early media with DTLS-SRTP is an ongoing work item for the IETF rtcweb and other WGs[4].

As the audio or video communication starts, the browser has to provide the relevant information into the media path. The browser mediates the access that the media path has to the local system used. The browser API also can trigger another API called Media Capture [16]. This API is able to control several features of the browser machine, namely microphone and camera(s) to provide the audio and video content. In a realistic web application scenario, this goes way beyond the mere establishment of a communication. This also explains the significant attention paid in the IETF WebRTC security drafts to topics like identity management, since the users' decisions about permissions for sharing their device's audio and video will be highly dependent on each user's perceptions of who is on the other end of a call and need to be independent for each user.

The JS contained in the page that controls the WebRTC sessions can also access and control many other resources that are important in a bilateral or multilateral audio or video communication. This includes things like accelerometer data, or ambient light informaion, and perhaps in future, address books, location, temperature and the many other sensors a smartphone may make available. It is therefore of prime interest to control how far information accessed by one part of a web application can be delivered to another part. Already today, there are social networks offering audio and video communication within their own context. But this is irretrievably entangled with their own identity management systems. Regularly, we hear that such social networking applications escape their limitations and send local data over the wires, see for example a Dutch report on WhatsApp [2].

The WebRTC JS module specifications addressed by the W3C contains a set of object definitions and function calls for the following features:

- browser-to-browser connections
- browser-to-browser Data API
- browser-to-browser Dual-Tone Multi-Frequency (DTMF) signalling
- Identity Provider Interaction
- Various extension points

The main items of note here are the MediaStream and PeerConnection objects and APIs. The former represents the stream(s) of media data involved in a call that are supported by both browsers, the negotiation of the details of which is one of the main complexities of WebRTC. The latter (PeerConnection) provides methods for two browsers to communicate data dealing with the many problematic connectivity issues that can arise.

With these function calls a WebRTC application can be programmed using JS. This web application comes embedded in the HTML5 page from the server and can make network connections of all kinds within its security limitations that are mainly determined by the Same Origin Policy (SOP)[5].

There is a potential discrepancy between what a web application is able to do and what a good web application ought to do. One of the hardest and most onerous achievements in the Device API Working Group was precisely to get to agreement on such Best Practices, which for WebRTC have yet to emerge. The Device API and WebRTC WGs have formed a task force to define the *Media Capture and Streams* capability [50] that will mediate WebRTC access to devices and manipulation of data streams from those devices but this is still work in progress at the time of writing[6].

[4] See [18] for a treatment in SIP.
[5] See D1.1 for explanations about SOP limitations.
[6] `http://w3c.github.io/mediacapture-main/getusermedia.html`

1.2.2 Core WebRTC Specifications

1. WebRTC 1.0: Real-Time Communication Between Browsers [10] W3C Working Draft 10 September 2013
2. Media Capture and Streams [15] W3C Working Draft 03 September 2013
3. MediaStream Capture Scenarios [50] W3C Working Draft 06 March 2012
4. MediaStream Recording [7] W3C Working Draft 05 February 2013
5. Mediastream Image Capture [52] W3C First Public Working Draft 09 July 2013

1.2.3 Other relevant Documents

1. Standards for Web Applications on Mobile: current state and roadmap [36]
2. Device API Access Control Use Cases and Requirements [5] W3C Working Group Note 17 March 2011
3. Device API Privacy Requirements [20] W3C Working Group Note 29 June 2010
4. Web Application Privacy Best Practices [37]W3C Working Group Note 03 July 2012
5. Web Intents [13] W3C Working Group Note 23 May 2013

1.3 Implementations

The WebRTC project[7] is an open-source project supported by Google, Mozilla and Opera. It reports implementation of current drafts in all the three browsers to a certain extent.

For Google Chrome, both getUserMedia and PeerConnection are implemented and shipping in the desktop version of Chrome for Windows, Linux and Mac. These APIs do not require any flags or command line switches to use as they are now part of Chrome Stable.

Mozilla Firefox has implemented getUserMedia, PeerConnection and DataChannels in the desktop version of Firefox for Windows, Linux and Mac. getUserMedia does not require any flags to use in Firefox 20 or later. It has not implemented the Recording API yet, but Mozilla plans to support it in future versions of Firefox.

For the moment, both browsers still use prefixed function names in their implementation[8]:

W3C Standard	Chrome	Firefox
getUserMedia	webkitGetUserMedia	mozGetUserMedia
RTCPeerConnection	webkitRTCPeerConnection	mozRTCPeerConnection
RTCSessionDescription	RTCSessionDescription	mozRTCSessionDescription
RTCIceCandidate	RTCIceCandidate	mozRTCIceCandidate

It is already a common experience in W3C that once the specifications are more mature, the number of implementations will grow and the implementations will diversify. The fact that two of the main browsers with a market reach of over 50% are implementing WebRTC means that there is a high probability that the end user will get widespread access to the technology within a reasonable time.

[7] `http://www.webrtc.org`
[8] `http://www.webrtc.org/interop`

1.4 Legalities

Telephony and, more generally, inter-personal messaging of all kinds hold a special place in our societies. Nearly all constitutions and charters of fundamental rights protect the communication channels we use. WebRTC thus has to respond to the criteria given in Article 7 of the Charter of Fundamental Rights of the European Union [57] stating:

> "*Everyone has the right to respect for his or her private and family life, home and communications*".

These rights guaranteed in Article 7 correspond to those guaranteed by Article 8 of the ECHR [1]. To take account of developments in technology the word "correspondence" has been replaced by "communications". The nuanced change in the wording means that it is conceivable that some jurisdictions might conclude that these rights to respect for communications privacy and security apply not just to 'traditional' communications technology but also to newer technology such as WebRTC. This in turn might mean that one could apply the rich case law developed by the European Court of Human Rights(ECHR) when considering WebRTC.

The Pervasive Monitoring (PM) many of us suspected and its confirmation by Edward Snowden show the significant impact that insecure real time communications can ultimately have. The ECHR has developed the rule that intervention "is legitimate and may become necessary in a democratic society ... for the protecting ... of the rights and freedoms of others. It confers a positive obligation of the governmental actors to protect the rights of individuals" [56] (page 79). In its communication COM(2010)573 "Strategy for the effective implementation of the Charter of Fundamental Rights by the European Union"[9], the European Commission sets ambitious objectives for its policy following the entry into force of the Lisbon Treaty. The Commission wants to make the fundamental rights provided for in the Charter of Fundamental Rights as effective as possible. This implies a duty for the European Commission to act in securing its Citizens' communications. This report also tries to give hints as to what the actions of the European Commission could be.

On 8 April 2014, the European Court of Justice (ECJ)[10] declared the Data Retention Directive 2006/24/EC invalid. The Directive, specifying a 6 month long full retention of all traffic data, exceeded the limits imposed by compliance with the principle of proportionality in the light of Articles 7, 8 and 52(1) of the Charter of Fundamental Rights of the European Union. In this decision, the court hints at requirements for a lawful processing of logged data.

There are limits to data retention as such, but the ECJ also erects further requirements concerning security of the resulting data.

- The court asked for access control to limit data access and use to what is strictly necessary in the light of the objective pursued.
- The court encourages governments to make time of data retention dependent on the category of data collected
- For future regulations in this area, the court requires the making of rules concerning the security of data retained by providers that are specific and adapted to (i) the vast quantity of data whose retention is required by that directive, (ii) the sensitive nature of that data and (iii) the risk of unlawful access to that data; rules which would serve, in particular, to govern the protection and security of the data in question in a clear and strict manner in order to ensure their full integrity and confidentiality.
- Data retention, furthermore, needs to take technical and organisational measures into account and must ensure the irreversible destruction of the data at the end of the retention period.

[9] `http://ec.europa.eu/justice/news/intro/doc/com_2010_573_en.pdf`
[10] Joined Cases C-293/12 and C-594/12

This ruling could have some impact on WebRTC – if it requires new national legislation in various European countries, then the extent to which WebRTC traffic is included or excluded in any future data retention laws may have a significant impact on deployments, though perhaps less of an impact on protocols. The "danger" being that requirements for data retention, if they include WebRTC call data or metadata, could act as a disincentive to many web sites from deploying WebRTC. However, it is too early to tell if such changes will or will not impact on WebRTC; for now, we merely note the risk here.

Chapter 2

WebRTC Assets

In this chapter we begin the analysis of the WebRTC framework following the methodology from STREWS Web-platform Security Guide (Deliveable D1.1 [25]). This analysis is of course a work-in-progress and will continue to be such so long as the work of the IETF rtcweb, W3C webrtc working groups and of implementers and deployments is similarly a work-in-progress. However, it is useful to try do the analysis in parallel with those developments. (That being one of the main hypotheses behind STREWS.) In this chapter (and overall in the remainder of the document) we emphasise the new aspects that WebRTC adds to the existing Web platform and do not for example, consider the generic Web platform security issues that were already analysed previously.

2.1 Enumeration

We can identify the following assets related to WebRTC.

1. Browser - the end user's browser, which the end-user expects to be operating, safely, on behalf of the end-user and not, for example, working solely to the benefit of any web site accessed. This is a user asset and implements the W3C WebRTC APIs, as well as other normal browser functions. The browser is also as usual responsible for enforcing Web and WebRTC security controls, for example, the Same Origin Policy (SOP). For WebRTC the new APIs offered by the browser and how those interact with the SOP is one of our main concerns.

2. Client Machine - the end user's computer (or phone) "outside" the browser together with the potentially sensitive devices that the machine is fitted with (e.g., microphone, camera(s), file system and screen). This is relevant to the WebRTC as some features (in particular screen sharing) could impact on other applications running on the host, or on the host operating system itself. This is considered a user asset here, though of course could also be a corporate asset in another view. Our main interest here is how the new WebRTC functions provide potential new targets for attack via the Web.

3. Server Machine (aka Web application) - the application on the web server that sets up and manages the call. In this context, since the WebRTC is really a set of applications running on a web server, we mainly treat the web server as an application asset, but all the usual web server vulnerablities remain relevant (e.g., direct access to storage etc.). Our main interest here is in how the web server could be attacked via the new WebRTC application. Note that there can be two web servers involved in which case there is signalling needed between the web servers which is not standardised by WebRTC.

4. Client-side Application Code (WebRTC JS) - the JavaScript that manages the calls on the user's browser, that is disrtibuted from the web server. This includes JavaScript libraries that may be downloaded from the web server or a third party web server and that implement parts of the WebRTC JS functionality. The difference here is that these libraries are not usually under the control of the WebRTC JS author. This is an application asset.

5. Identify provider's infrastrucuture (IdP) - this is actually quite a complex asset, but for the purposes of this analysis can be regarded as a single service. Each user on a call may of course use a different IdP, and is likely to use the IdP for much more than WebRTC purposes. This is an infrastructure asset.

6. STUN/TURN server - the server(s) that operate the STUN/TURN services used for ICE. This is an infrastructure asset.

For the current study, we ignore gateways to "legacy" services such as conference bridges or SIP-based VoIP services and the PSTN.

Figure 2.1: WebRTC Assets and Flows

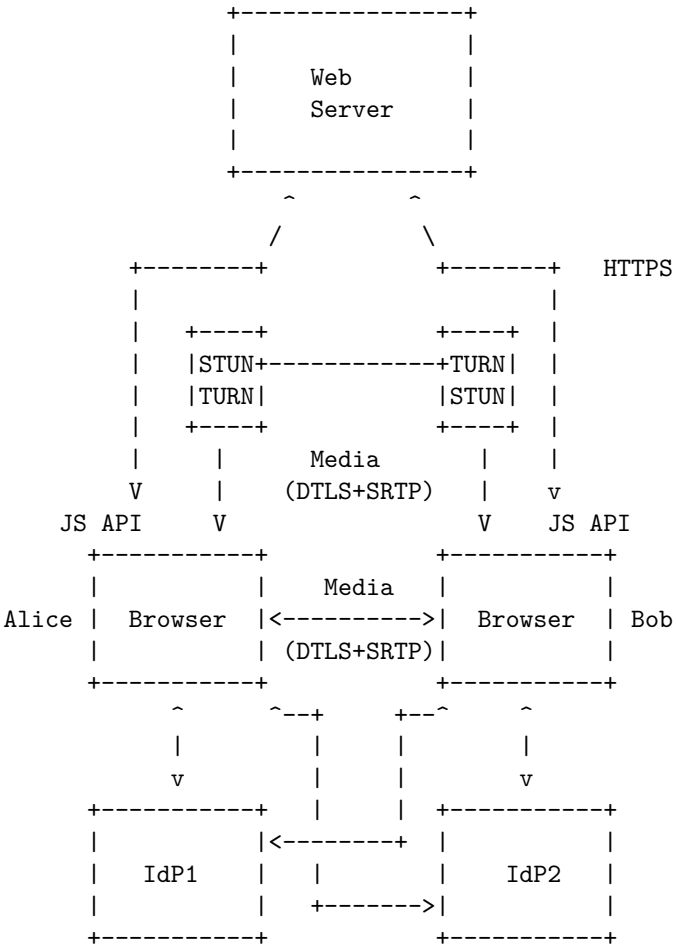

Figure 2.1 is a modified version of Figure 3 from [63] that shows relationships and some flows that can occur between these assets. Note that the Media could flow either directly between two browsers or via TURN servers, and may involve STUN servers as part of the ICE algorithm.

Figure 2.2: WebRTC inside the Browser

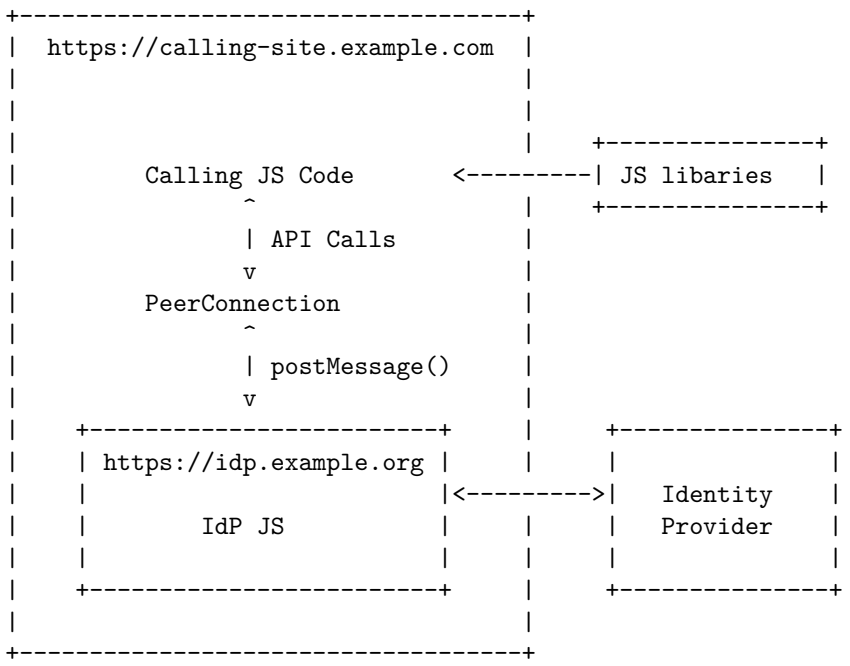

Figure 2.1 (based on the figure from Section 5.6.2 of [63]) shows more detail of the assets within the browser. The PeerConnection API is an example of the WebRTC APIs offered in the broswer.

2.2 New WebRTC Interactions raise new avenues for Threats

Based on an architectural assessment of the WebRTC architecture sketched in Figure 2.1, the following security have been identified:

1. A browser can now attack another browser directly. This is new to the Web model that normally works in a client-server environment.
2. A Web server can attack a browser via another browser. If there are two web servers involved in a call, then each of those can cause "their" browser to attack the other party in the call. This is really consequence of the previous case in an indirect manifestation, as Alice's browser is executing code from Alice's web server whilst talking to Bob's browser.
3. If there are two web servers involved in a call, then each of those can attack the other via whatever signalling is in place between them.
4. The STUN/TURN servers (particularly if TURN is used) can attack the communications between the browsers in various ways. Note that the STUN/TURN server can be selected by, but will often not be operated by, the web server and can also be discovered algorithmically via ICE.
5. Browsers and web servers can attack STUN/TURN servers, for example to consume (unauthorized) bandwidth, or as a potential denial of service (DoS).
6. Web servers can monitor calling patterns and metadata.
7. STUN/TURN servers can monitor calling patterns and metadata.

8. New technologies (new to the Web) such as SRTP, SDP, codecs etc. will have inevitable implementation bugs opening up attack vectors.

9. Call metadata which will be accumulated in logs of the various components will contain information about pairs of (or more) users that are not affiliated with the same web site. Previously, the Web essentially allowed accumulation of similar metadata only when two users used the same web site, absent which the logged data represented how a single user interacted with that and other web sites.

10. The semi-standardised API to allow interaction with the IdP provides a new way for attackers to attempt phishing attacks.

11. Assumptions previously made about voice communications being somewhat "out-of-band" of the Web are invalidated, especially on mobile phones. This is more of a layer 8 security consideration, rather then technically part of WebRTC but significantly broader deployment of WebRTC compared to VoIP solutions may mean it becomes significant.

12. Screensharing is a common conferencing tool that will be a new browser feature. In principle, while a screen is being shared, malicious JS could continually shapshot the image and exfiltrate that to a command and control server once the user has once granted permission for screen sharing for that origin.

13. Although we don't examine it further here, the interactions between the web and legacy signalled communications (such as conference calling systems and the PSTN) could expose both to the other, resulting in damage.

2.3 New Threats by Asset

In contrast to the current Web platform, we do not yet necessarily have crisp names and definitions for a number of the threats below. As happened in the case of other new technolgies, we expect it will take some time until the terminology for and etymology of the more important threats becomes well established.

In the following we generally will not repeat descriptions of threats that are already described in the STREWS Web-platform Security Guide (Deliveable D1.1, [25]).

2.3.1 Browser

We have two levels of vulnerabilities here. First there is the WebRTC JS itself that can be used to gain access to functionality of the Client Machine or browser that is supposed to be disallowed, or that might go beyond the users wishes.

The second level is a rogue web application that tries to escape the browser sandbox to access the operating system of the client machine or store malicious code on it so that an attacker can control the client machine from outside.

Escape Client-side Sandbox/Environment

The WebRTC JS can try to escape the sandbox the browser is providing and access local storage objects or browser functions. Ultimately, the loaded code can try to escape from the browser and install malicious code on the client side that can be used to take over the entire machine (root access, or the equivalent in the case of, for example a Chromebook machine[1]).

There are many types of attacks of this type reported. People are lured into clicking on some object that is malware. As the JS of the web application also controls the dynamic availability

[1] `http://www.google.com/intl/en_uk/chrome/devices/chromebooks.html`

of links, one could imagine that a WebRTC application does the same by loading malware on the client computer.

In particular, if a browser implements screen sharing via a browser extension (and not via "normal" JS) then that extension will have greater power to, for example, ignore the SOP as compared with normal JS. At least one browser implementer (Chrome) at present plans to support screen sharing via this mechanism. Reportedly, this is being done in order to raise-the-bar of security, since extensions (known as plugins in other browsers) must be deliberately installed, however that also gives the extension (and hence possibly malicious code) enhanced powers, outside the sandbox. There is also the issue here that if different browsers implement screen sharing differently then different security behaviours will be seen in each.

SDP triggered bugs

CVE-2013-0843[2] is a buffer size vs. sample rate bug that could be exploited via SDP from malicious web sites.

New code with old bugs

As WebRTC requires significant new code to be included in the browser, and as implementers will find that code in various libraries, and as it is likely that none of those libraries will have been as tested to the extent that browers are or in conjunction with all the browser (versions) that the library might be used with, we can predict a range of vulnerabilities will continue tobe found in such code for some time to come. For example, CVE-2013-6631[3] is a use-after-free bug in a codec

Many such vulnerabilities can be expected.

Client-side Content Storage

Recently introduced capabilities in the browser allow a web application to store data at the client-side, within the browser. After a successful negotiation using the WebRTC API and the JSEP protocol and the SDP descriptions, a browser or the WebRTC JS may chose to store the result of the negotiation into local storage. Gaining access to this storage may allow an attacker to manipulate the stored descriptions, which typically includes high value information such as contact information for peers contacted via a particular origin. This could be used to trick the user into communicating with the wrong peer, which is perhaps mostly a nuisance, but more seriously could leak the user's communications history. Any web site offering WebRTC functionality that has for example an XSS vulnerability can offer a conduit for exploitation of this threat.

Probing Browser Capabilities

By making many attempts to connect to a browser with various different kinds of SDP proposals, an attacker can determine browser capabilities, preferences and other information about the user. Similarly, a badly designed WebRTC web application can turn a smartphone into a perfect tracking device. Inappropriate access to sensors and the measurements made by them are problems of the permission system that are analysed in Chapter 3. Such access also creates new possibilities for an advanced fingerprinting system.

Permission Management Abuse

Handing normal device access permissions (for example, microphone, camera) for WebRTC may prove difficult. (See also Chapter 3. In particular, users are unlikely to remember (or understand)

[2] `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-0843`
[3] `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-6631`

to which origins they have granted which permissions. And most browsers discourage resetting such permissions because they only allow that to be done at a very coarse granularity (typically "forget all site data for this site"). Over time, for many users, this is likely to lead to a situation where the most popular sites have been granted permissions that would allow them to eavesdrop on the user. And for some of those sites, for some durations, such eavesdropping will be done, either due to rogue employees, or related to some form of advertising, or via some exploit code of bug in a JS library used by the site.

New WebRTC Interstitials

Experience seems to show that any new security relevant interstitial dialogs will at some point cause problems. CVE-2014-1499[4] is an early demonstration of this, based on timing and such a dialog, with the former under JS control.

2.3.2 Client Machine

Access Camera or Microphone

WebRTC requires controlled access to the camera and/or microphone in order to originate data streams. Some permission systems, on smartphones in particular, are not granular enough. In Chapter 3 we provide detailed analysis of possible attacks here. For example, a WebRTC application on an Android based system may ask for permission to activate the camera. But it does not control the circumstances and timing of camera activation. This means an attacker who gains control over the browser may gain control over camera, microphone and other sensors in a smartphone whenever the browser or other application is running. This turns the device into a perfect eavesdropping system that may also provide GPS coordinates.

Access Devices

Another very important aspect is accessing hardware sensors from the web application. This is mainly dealt with in the Device API Working Group in W3C[5]. See also Section 3 regarding permissions to access those sensors. This covers also the most prominent fears around secret activation of camera and microphone. As an example, fingerprint sensor outputs, if raw data is accessible, could also be stolen and could allow other kinds of login.

Screen Sharing Snapshots

As previously noted, once screen sharing is enabled this enables many threats. For example, the ability to take a snapshot of the screen every 10ms say, would defeat any second-factor authentication scheme that depended on selecting known images and schemes that specify subsets of password characters. Highly sensitive application data, such as bank balances or transfer details, or healthcare information could also be captured this way.

Call Recording

Some browsers may (via plugins, extensions or natively) provide call recording features. While recording ought be known to both parties, this will not always be the case and many calls (both voice and video) will be recorded without all appropriate permissions having been acquired.

[4] http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-1499
[5] http://www.w3.org/2009/dap/

Caller-Callee Mismatch vs. Same Origin Policy

The basis for current browser security is the Same Origin Policy (SOP), and that is being extended for use with WebRTC. For example, browsers will be allowed to remember permission settings based on the SOP. This however does not match the caller/callee model that users might find more intuitive. For example, I may be much more likely to want to receive a call from my friend, rather than allow in inbound call that is setup by example.com. One could predict that such mismatches will cause some security and privacy issues as WebRTC develops.

2.3.3 Server Machine

Call Man in the Middle

The server machine does not, in the nominal case, have access to the content of the real-time communication. But as the Web server controls all the signalling, it can manipulate the data channel to go to the wrong calling party (i.e. not Bob) thus allowing for an easy man in the middle (MITM) attack. For example, setting up a call as a conference call. In any case, many calls will be between a browser and a server (e.g. customer support) and so can easily be recorded on the server side.

Call Recording

When the server is in the media path, unauthorised recordings may be made, as in the browser case.

Directory access

The Web application containing the real-time communication will have access to a directory of users of the server or other Web servers that offer real-time communication capabilities. This is not really gaining full access to a Server machine (as in root prompt), but a Web application could be manipulated on the client side to allow for bulk access to, or probing of, such directory information. A recent news report of the attempt by Facebook to acquire WhatsApp[6] for $19 billion have been translated into cost per user or cost per address book. In the deal between Facebook and WhatsApp, newspapers talked about $45 per person. WhatsApp forces people to upload their entire address book including all the phone numbers. This means by acquiring WhatsApp, Facebook acquires a lot of user data. This results in the extremely high valuation of the proposed purchase.

In our context, it means that the directory information on the server is potentially very valuable to attackers, especially if one can download the entire directory. So while there is no abuse of Server-side Privileges in a classic sense, a Web application may be constructed or manipulated in a way to allow it to gain access to very valuable directory information. Such directory information might be a target of and exploitable via “normal” web attack such as SQL injection.

Given that we have also a distributed scenario with a second Web server, an attacker could gain access to more than one directory. If we have a world directory (as, for example, we have for DNS), access to one rogue Web application with unlimited access to all directories could turn out to be a major privacy breach that may even trigger legal action or data breach notification obligations. Further considerations concerning leaking information are found in Section 7.3

Abuse Server-side Application Privileges

The privileges of a WebRTC client on the server are rather low. As we have seen, the server controls all signalling. This means that even if an attacker would be able to get control over a

[6]The Register 20 February 2014, `http://www.theregister.co.uk/2014/02/20/facebook_whatsapp_19bn_buy_also_45_for_your_phonebook/`

browser application, this attacker has not gained a much better position to attack the server or to manipulate content on the server.

Of course an application could have access to some service like an online gaming platform that requires payment. Gaining access to the application allows the attacker to use those credentials for himself. It may be interesting to see if the communication credentials for the online gaming platform are the same as credentials to actually play the games offered. Mitigation may involve the use of authentication systems that do not use the same credentials for all privileges.

Directly Access Server-side Storage

As WebRTC involves Browser-to-Browser communication, the WebRTC server only stores information about possible or potential connections between browsers. The most valuable information stored in the web server is probably the state of the connection.

Directly accessing the server-side storage will allow an attacker to find out about other potential participants of a communication and the current state of that communication. But as the credentials are exchanged in the SDP exchanges between the browsers trying to establish a direct real time communication, the direct access to the server side storage is only done to access or influence the control channel that controls the signalling of the Browser-to-Browser communication.

Consume Server Resources

WebRTC calls involve some server resources, in particular those that involve a server-side callee (e.g., calls to a helpdesk). Clients or a botnet could use WebRTC as a significant DoS accelerator in some cases, for example if it were possible to direct video traffic from a large set of callers at an endpoint within the server infrastructure.

2.3.4 Client Side Application Code

Attack on Web Server

Because of the complexity of the WebRTC platform, it is highly probable that certain WebRTC providers will offer ready to insert JS code for pages that want to use their services. As the Web server only needs to provide the signalling, the amount of executable code on the server side is rather limited. Thus we can conclude here that running Attacker-controlled Server-side components to attack the Server machine is not very likely. This form of attack is more likely to be used against the client machine.

One can of course expect there to be bugs and possibly backdoors in such shared code, and vulnerabilities due to the complexity of WebRTC.

Attack on local Browser

Similar to the above, one can expect vulnerabilities in library code to create exploit opportunities against the caller's browser. This could come in the form of a library or piece of WebRTC code that exposes the browser to remote access, or that allows for additional fingerprinting of the browser (e.g. if a library calls home to its author) that could in turn be used for tracking or as a prelude to other attacks based on the fingerprint.

Attack on remote Browser

Similar to the above, one can expect vulnerabilities in library code to create exploit opportunities against the peer's browser.

Attack on Web server

A JS library used (included/imported) by the WebRTC JS code by a web site could be used to attack that web site.

2.3.5 Identity Provider

Phishing

The fact that the WebRTC APIs do not include a "special" IdP API, means that it is not possible for browser "chrome" to be used to help users detect the differences between their interactions with real IdPs compared to with a fake site masquerading as an IdP. This therefore will not make phishing harder and may even make it easier, if WebRTC results in users becoming accustomed to entering their IdP credentials after visiting a web site and clicking on a link that sets up a WebRTC call.

Tracking

The ways in which WebRTC allows IdP's to control the duration of DTLS sessions for calls (via issuing credentials) means that an IdP that issues very short lived credentials would be in a good position to track a user's calls.

Authentication Credentials

As already laid out above, many of the use cases around WebRTC also include payment and subscription models. Those subscriptions are tied to credentials of all sorts that are more or less secure. An attacker who can get hold of such credentials thus gets a free ride on the subscription service while the victim will have to pay.

WebRTC further entrenches massive central IdPs

Web sites offering inter-user calling services will need some form of user identification. For some sites (e.g., bulletin boards) with existing nicknames or user naming, that may be relatively simple. However, for sites without such identification, WebRTC is likely to further entrench the use of massive centralised IdPs. Creating such large targets means that breaches they suffer have greater impact and also makes them more attactive to attack.

2.3.6 STUN/TURN Server

STUN/TURN Monitor

Assuming that DTLS cannot be compromised, then attacks on, or controlling, or from, the STUN/TURN server can mostly be used for monitoring, since, especially with TURN, all call meta-data is available to the TURN server.

If we assume that the economics of the situation will lead towards larger scale STUN/TURN servers being operated in order to leverage call meta-data for marketing purposes, then such attacks may become relatively high probability, at least for some calls.

DTLS Confusion

If DTLS can in fact be worked around, for example if WebRTC Browser clients made it easy for a user to click "ok to accept" as a way to bypass DTLS server certificate authentication, or if DTLS keying is entirely based on uncertified long term or ephemeral public keys (even if names are vouched for by some Identtiy Provider) then TURN servers in particular represent a fine place from which to launch a successful MITM attack on call data.

DoS STUN/TURN

Given the the co-ordinates of (and "password" for) the STUN/TURN servers will be downloaded to clients in JS code, one can expect that that information quickly becomes well known. This is likely to make such servers an attactive target for DoS attacks.

Other People's Bandwidth

Given that TURN servers in particular will be seen as the source of packets that consume inter-AS bandwidth, it may well be very attractive for browsers and web applications to attempt to consume bandwidth being paid for by others simply by preferring particular TURN servers.

2.4 Threats against Content in Transit

Content in transit is always vulnerable, especially on a packet switched network where all middle boxes have access to the information unless the payload in the packets is encrypted. Note that even if the payload of the packets is encrypted, some headers necessarily remain visible to the boxes guiding the packets through the network. Unless one uses Onion Routing [28] or some equivalent, source and destination are clearly visible. As Ed Felten noted: "Metadata can often be a proxy for content" [14].

But this is a generic issue that is not special to WebRTC and is being addressed by the IETF [31] in the future[7]. For the analysis, we do not take this generic and inherent possibility of wiretapping and eavesdropping of the Internet into account. For WebRTC, eavesdropping of media content from the network should be greatly mitigated by the use of transport layer security for all network connections.

2.4.1 Inject and Manipulate Traffic

The exploit described here has apparently been used by the NSA in the "*Quantum*"[8] program. To trick targets into visiting a FoxAcid server, the NSA relies on its secret partnerships with US telecoms companies. As part of the Turmoil system, the NSA places secret servers, codenamed Quantum, at key places on the internet backbone. This placement ensures that they can react faster than other websites can. By exploiting that speed difference, these servers can impersonate a visited website to the target before the legitimate website can respond, thereby tricking the target's browser to visit a FoxAcid server.

From an attacker's point of view being able to control the content that is fed into the browsers API means it can control the entire communication between the the two users via their browsers. This in turn means that control over the server side of a WebRTC communication allows for a range of manipulations of the data channel between the two communicating browsers.

As we have seen, by controlling the Web server and sending a malicious WebRTC application to the browser, the attacker gains access to most of the valuable information one can obtain in a WebRTC scenario. Not controlling the (well protected) web server, a MITM attack allows an attacker to remove the legitimate web application and to send malicious code instead. This makes it crucial for WebRTC that the Web server is trusted not to spawn malicious code and that TLS/HTTPs is used for the interaction between browser and Web server.

But the injection can not only happen by sending malicious code. As the Web server controls the signalling of the Real-time communication, altering those messages also gives control over

[7] STRINT Workshop, A W3C/IAB workshop on Strengthening the Internet Against Pervasive Monitoring (STRINT), `https://www.w3.org/2014/strint/`, especially draft-crocker-strint-workshop-messaging [22]

[8] Wired, 13 November 2013 `http://www.wired.com/opinion/2013/11/this-is-how-the-internet-backbone-has-been-turned-into-a-weapon/` citing The Guardian, 4 October 2013 `http://www.theguardian.com/world/2013/oct/04/tor-attacks-nsa-users-online-anonymity`

the entire communication. And this may not only happen in the relation between the Web server and the browser. In the more complex scenario of 1.2 with at least two distinct services talking to each other, the Web server to Web server communication can be also a target for manipulation of content in transit. The NSA used this to intercept traffic between data centres of Internet giants. While they had the traffic between browser and their servers encrypted, the traffic between their data centres wasn't and allowed clear access to everything[9].

2.4.2 Generate Traffic

Denial of service attacks are possible against the Web server providing the WebRTC service as well as against the browser. One could imagine that heavy SDP traffic from a botnet against a single browser can shut this machine down.

As WebRTC can use SIP and XMPP for the signalling, all the DDoS characteristics apply. As in politics, calling campaigns, mainly to governments or parliaments, are already current practice, it is worth noting that those will be greatly facilitated. A NGO could set up a page that uses a certain WebRTC service and then propagate the URI via social networks. Someone clicking on the link would automatically call a certain connection in parliaments or governments.

In a report from 2008[10] the risk of SIP DDoS is commented in a rather laconic way. It is seen as one way to abuse diagnostic tools to mute a service or to achieve a buffer overflow.

2.4.3 Authenticated Session

Use cases for WebRTC include telephony that the user has to pay for. Another use case talks about online video games that typically require some subscription and payment. The same applies to the video on demand services that require a subscription or a payment. Paid services then require authentication and credentials. Rather than leverage the IdP infrastructure, for some web sites the relevant credentials might be exchanged using SDP over the signalling channel where eavesdropping can happen. Thus an attacker can hijack a session and get a free ride on the cost of the victim whose credentials the attacker is using. The actual attack is mitigated by the use of TLS on the signalling channel as was described above.

2.4.4 Forge Request

It is clear that once the browser is on a page containing JS, this JS may mimic a WebRTC client to abuse the connection to test out some other targeted client machine. It can now, in the name of the hijacked browser request SDP information and thus explore the capabilities of the target. A forged request could also be used to escape subscription fees.

2.4.5 Personal Information

As said already many times, the biggest threats to the user's privacy via the network might be:

1. eavesdropping on the browser-to-browser communication
2. bulk access and abuse of information from the WebRTC directory
3. profile building by the Web server

Eavesdropping on the browser-to-browser communication is rather hard on the middle point of the communications, as the line between the browsers is encrypted. But the endpoints are weak. A malicious WebRTC web application could easily re-route the entire communication to a third party.

[9] Soltani, Gellman, NSA infiltrates links to Yahoo, Google data centers worldwide, Snowden documents say [74]
[10] Jose Nazario, Global SIP Attack Activity `http://www.arbornetworks.com/asert/2008/10/global-sip-attack-activity/` last seen 2014-02-24

As we have seen with the whois service and other directory information on the web, those are collected and abused for profile building and unsolicited commercial communications. For WebRTC there is an additional risk that we know from the plain old telephone system. In the US in the early 2002, the amount of unsolicited commercial calls on the average American household was such that people started to disconnect their phone. The phone became nearly unusable. An initiative from the Bush administration and the FTC ended in a so called do-not-call list. All consumers can subscribe their number to this list and commercial calls are excluded. Either WebRTC will have to participate in this programme or it will generate the same amount of trouble for people online and thus hinder the broad adoption of WebRTC.

Finally, WebRTC will be used by the giants of social networking and social media. They typically make money by profiling people and selling those profiles to retail and other commercial entities for better targeting or for risk minimisation. In order to find people to communicate with, social networking is an ideal directory services. This will increase the pressure on people to participate in large social networking providers because of the increases in the economic networking effects of their users base. If a central TURN server is added to the picture, information concentration is increased and will attract more actors for pervasive monitoring. The data obtained is used for highly diverse purposes. Governments may use it to put the user on a no-fly list. Companies may use it to inflict targeted advertisement and retargeting on people for the use of their services.

Part II

Areas for In-depth Investigation

In this section we delve into more detail in areas where STREWS can potentially contribute to the development of WebRTC based on the limited effort dedicated to this case study.

Note that by "in-depth" we do not mean that STREWS has finished an exhaustive treatment of these topics, but rather that we are continuing to monitor them and plan to interact with the research and standardisation communities on these topics on an ongoing basis. How well (or badly) that works out in practice will be one of the more interesting results of the case-study approach being taken here.

Chapter 3

Permissions

In this section we discuss how the WebRTC model interacts with user handling of permissions at the browser.

3.1 General considerations

A WebRTC session that uses video or audio requires access to the camera and microphone. The browser or run-time environment does not automatically give a downloaded application access to those. The access is subject to permissions, which are granted by the user.

Permissions can be given for a session, an application, an application server, a peer, a period of time, or some combination of those. In the case of an application, it can be given to a certain version only, or to all versions.

A permission model that is too fine-grained has the danger that the user gets annoyed at the number of times he has to grant permission, or that the user does not understand the permission being requested (as is quite common with e.g. Android). He might stop reading the questions and just click OK.

A permission model that is too coarse may result in permissions being given for things the user didn't expect or didn't want. E.g., a user might give permission to a certain application to use the camera, thinking that the permission is only for that application, while it is in fact for all applications from the same server.

The system's model for permissions should match the user's mental model, and be clearly explained, otherwise the user could easily make wrong decisions.

An example of a possible mismatch between the user's model and the system's model is the *identity* of the resource that a permission is attached to. In the case that the permission to use the camera is given to an application server, it is (probably) attached internally to a domain name. If the DNS server is compromised, the permission is thus attached to the wrong server (unless the connection to the server is protected by TLS and a mismatch in the TLS certificate gives the attack away).

Users should thus also understand how well the identity is verified. Is there a cryptographic signature that was verified and if so, by whom: your browser or some third party? And in the latter case, has the identity of that third party itself been verified? In the standard WebRTC model, permissions that are not backed by TLS (server) origin authentication are only per-session. In order for a site to be given long-lasting permission to access the camera or microphone, the site has to be authenticated.

While this binding of device permissions to web server origin authentication seems like a reasonable starting point, it is not clear whether that mapping will prove robust or problematic over time.

Permissions should also be able to be revoked, especially if they are long-running or permanent. And it should be easy for the user to revoke a permission. (Which requires being able to inspect which permissions have been granted.)

3.2 State of current implementations

The two implementations of WebRTC that exist at the time of writing (early 2014, this section was not updated in September) (see Section 1.3) implement two similar, but subtly different models of permissions.

- **Chrome:** The Chrome browser asks permissions for using the camera (which implies use of the microphone as well) when an application is downloaded that requires the camera. That permission holds for all applications from the same server (the same domain name, protocol and port). The duration of the permission depends on the protocol used to download the application: if the application came over an HTTP link, the permission lasts only as long as the browser is running. But if the application came over HTTPS, the permission is permanent. An icon near the location bar indicates that the currently running application has permission to use the camera. Permissions can be revoked by clicking on the icon and opening "media settings" in the dialog that appears.
- **Firefox:** Firefox asks permission for the camera and the microphone (in this case they can be granted separately, if desired) in a similar way. The permission is again for all applications from a single server. But unlike in Chrome, all permissions are only for the duration of the session, that is, until the browser closes. There is no way to revoke a permission, except by closing the browser.

These two browsers, and Chrome especially, have clearly chosen to bother the user as little as possible. It is left to the applications themselves to provide more fine-grained permissions, such as which peers to send video to. Please refer to Table 3.1 [1] for a general overview on implemented permissions and the corresponding user interaction model in current Web browsers.

[1] This table is based on work by the HTML5Apps project. See `https://github.com/dontcallmedom/web-permissions-req/` for more.

Feature	Type	Time of request	Persistence	Visibility	Control on permission	Embeddability	denied signaling	Remarks
Geolocation	Prompt	At usage time	Session by default, optionally permanent	Chrome indicator	Via indicator	Always allowed, bound to origin of document calling script	Error callback with PERMISSION_DENIED code	(getCurrentPosition and watchPosition handled identically)
Direct camera access (getUserMedia)	Prompt	At usage time	Session by default, optionally permanent	Chrome indicator, pulsing reminder	Via indicator	Always allowed, bound to origin of document calling script	Error callback with PermissionDeniedError object	Can be combined with mike access. Also grants access to information about other available cameras
Direct mic access (getUserMedia)	Prompt	At usage time	Session by default, optionally permanent	Chrome indicator	Via indicator	Always allowed, bound to origin of document calling script	Error callback with PermissionDeniedError object	Can be combined with camera access. Also grants access to information about other available microphones
Get stuff from camera / mic (HTML Media Capture)	Implicit via user interaction	At usage time	One-off	N/A	N/A	Always allowed	No file object available	
Notification	Prompt	Upfront	Persistent	None (?)	?	Allowed	Notification.permission == "denied"	
Pop Up	Denied by default, Opt-in required	At usage time	One-off, optionally persistent	Chrome indicator	Via indicator	Allowed	window.open == null	
Fullscreen	Ask for forgiveness after "engagement gesture"	At usage time	One-off	Transient overlay	Through instructions	Allowed only with specific permission from parent (allowfullscreen attribute)	watch el.fullscreenenabled; fullscreenerror event if previously rejected	
Pointer Lock	Ask for forgiveness after "engagement gesture"	At usage time	One-off	Transient overlay	Through instructions	Allowed	document.pointerLockElement == null ; pointerlockchange event	
File picker	Implicit via user interaction	At usage time	One-off	N/A	N/A	Allowed	No file object	
Quota	Prompt	Upfront	Persistent	?	Convoluted	Allowed	Success (!) callback with StorageInfo.quota unchanged	

Table 3.1: Currently implemented permission and the respective user interaction model in Web browsers [35]

3.3 Potential security problems in the current implementations

The browsers only provide coarse permissions, but ask for them at the start of a specific application, which may lead the user to believe he has given a specific permission, when in fact he has given a wide-ranging permission.

The permissions are for a server, so if the same server offers another, malicious application and the user can be tricked into opening it, it will have access to the camera and microphone automatically.

The permissions are granted as soon as the application is loaded, that is, before there is a peer-to-peer connection, and they thus do not depend on who the user is going to talk to. If the connection is established and the other side isn't who the user expected, it is too late to deny the use of video.

In a desktop browser it is reasonably clear what a session is and how to end it: quitting the browser is the end of the session. But on a mobile device it may not be so clear when the browser is running or how to stop it. Thus the user may have stopped a Web app (closed the window or swiped away the "card"), but the browser may actually still be running and the permission is still in force. Thus if another application from the same application server is somehow started, it will have access to the camera and microphone, although the user thought the session was over.

In the case of Chrome, the permissions are permanent (if the application is downloaded over an HTTPS link) and thus are still in force when the user comes back later to use the application again, the application is no longer the same. The application might have evolved into something the user doesn't like, or the domain name may have changed hands and the application replaced by a very different one.

What the user also may not realize is that the application server can get a copy of the video and audio that is sent to the peer, if it wants to. The JS SOP allows the application to send information back to its origin server. There are no separate permissions for that. The application may even provide end-to-end encryption between the user and his peer, giving the user the strong impression that nobody can listen in, but the application server still can.

Permissions do not extend to what the peer can do with the video and audio. It is left to applications to establish protocols for communicating privacy settings to a peer.

Apart from live video and audio, WebRTC could also be used for screen sharing. The two existing implementations do not offer that feature at this time, so there are no permission models to compare yet.

Screen sharing can be read-only or read-write (the remote peer can generate mouse and keyboard events). Those could be separate permissions. Users might also want to grant permissions not to an application server or an application, but to a person. The permission could be for a session or permanent. And it might be useful to limit the screen sharing to (the windows of) one or two specific programs, so that the user can keep private information on the screen even while sharing some windows with a colleague or a help desk.

Screen-sharing is an attractive feature for developers of conferencing applications in particular, where it is a standard feature to allow e.g. presentation of slides to the conference session. However, its dangers in the WebRTC environment are only now being investigated. There is a potentially significant difference between this feature being a part of a proprietary application that is rarely run, and it being a standard built-in WebRTC feature.

However, screen sharing is inherently more problematic than access to camera and microphone, as once such a permission is granted then whenever the site is "active" (which may not be visible to the user), that site's JS can take a snapshot of the screen, read keystrokes etc. Thus, a customer support WebRTC session involving screen sharing could expose, for example, banking credentials in ways that users will find extremely counter-intuitive. For this reason, one implementation (Chrome) requires that screen sharing only be enabled after a plug-in has been

explicitly downloaded and installed. The logic here is that plug-ins for Chrome do not have to adhere to the SOP, and that since screen-sharing inherently violates the SOP, a user has to "approve" the non-SOP aspect via downloading the plug-in. Its is not clear if this distinction will be at all meaningful to users.

Chapter 4

Key management and binding identifiers to calls

WebRTC media security and "identity" depends on cryptographic key management. In this section we review (what is currently known/decided) on this topic.

First, a word on "identity" - it would be better if the WebRTC specifications and community dealt in terms of identifiers and avoided uses of the term identity. The reasons are legion and have been rehearsed elsewhere, for example in the mail community where many identifiers that share or involve different identities are involved in the processing of each message, for example, a mail message's "From" and "reply-to" header fields, which often differ from the SMTP protocol identifiers such as MAIL-FROM or the username used for authentication of a message submitter. (See RFC 5598 [21] for details.) In all of these cases the distinction between identifiers and identities enhances understanding and the accuracy of descriptions. However, the WebRTC community have not been making such distinctions, so we aim here for consistency with their terminology rather than attempt to "fix" what they are doing. Note though, that one might expect that as WebRTC matures, such distinctions come to the fore as deployment issues throw up different assumptions made by different actors as to when various identifiers can and cannot represent the same or different identities.

Aside from the underlying Web PKI, there are two main sets of keys to be managed for WebRTC, namely

1. STUN/TURN/ICE secrets.
2. DTLS keys.

There will also be secrets used in interactions between browsers and IdPs. We describe each of these separately and then consider the overview.

4.1 The Web PKI

The existing Web PKI, with its known flaws, (see also STREWS D1.1) underlies all WebRTC-specific key management. That means that browser "root" CA information (or trust points) are required and can essentially control all security. Today, it is an open question as to whether new developments in Web PKI (such as DANE [39] or Certificate Transparency [19]) will turn out to have sufficient utility to achieve deployment so for now we assume the current Web PKI model will also be used as the underpinning for WebRTC.

This means that the main problem with the Web PKI is inherited by WebRTC - that problem being that there is no effective scoping of the authority of a CA and any one of the hundreds

of CAs in the world can claim to be authoritative for the binding of any public key and any identifier.

Clearly the Web PKI is used to handle authentication of Web sites accessed via HTTPS URIs, but the same PKI also handles authentication of IdPs, which in many WebRTC use-cases will be responsible for handling caller identification.

4.2 User Authentication and Naming

User naming and authentication can be done in a site-specific manner or perhaps more commonly via an IdP.

Site-specific user naming and authentication is likely to vary significantly but will typically ultimately depend on a username and password and on forms-based web authentication. Clearly, compromise of weak user credentials such as these will carry forward into the potential for call client authentication and MITM failures.

Site-specific naming will be quite vulnerable to confusable names and squatting on names. However, such informal handles will be considered a feature for small sites.

IdP-based user authentication will also be ultimately based on username and password, though with the potential for second-factor authentication, for example, via a mechanism like HOBA [30] or perhaps more likely (though similar) based on mechanisms selected by the private FIDO alliance consortium[1].

IdP-based naming will be more likely vulnerable to domain confusability if one assumes that mainly large IdP's will garner market share.

Name confusabililty based attacks might become a vector for social engineering of various kinds. Given that the presentation of names will be under the control of the WebRTC application and not the browser chrome, one can expect that confusability could become a real issue, for some names at some sites, regardless of how careful the IdP may be.

MITM attacks based on compromised user credentials should be mostly theoretical, since the most common way to compromise such credentials is via database leakage, at which point the attacker can compromise the WebRTC application if a MITM is desired.

Spoofing attacks based on harvested passwords may become a significant attack given that the average user re-uses a password at roughly eight web sites. This will mainly represent a nuisance or for random harassment, one would expect calls authenticated in this way to be rare as a form of attack. However, such spoofed calls could become significant as a part of identity theft attacks, where the attacker is using the call authentication in order to acquire other information about the identity. For example, such a spoofed call could be useful in order to change the address of a payment cardholder.

4.3 ICE/STUN/TURN Credentials

Symmetric "secrets" are required in order to authenticate the STUN/TURN protocol messages used to establish the connection between the browsers as part of the ICE algorithm. Those messages may include a keep-alive message sent periodically in order to maintain NAT bindings for the call.

The basic model is for the web site to include a STUN/TURN server name (or address) and secret in the JS sent to the browser. That requires that the web site and STUN/TURN server share this secret. One can speculate about various deployment models and how those might impact on managing these keys but one can expect that such "secrets" will become widely known as they are in clear in the JS code.

It will likely be the case that the same STUN secret could be used for many web sites, if the STUN/TURN server is operated as a "free" service, if for no other reason than load balancing.

[1] `http://fidoalliance.org/`

The same secret is highly likely to be used by many browsers, so calling this a "secret" is somewhat of a misnomer, but this is used as a key and, in future, some standard automatic key management protocol could be developed for web-sites to acquire (sets of) STUN secrets to hand out to browsers wishing to make calls. There will likely be proprietary variants on this theme as well.

Another deployment model will need to work for enterprises, where the STUN/TURN server will either have to be accessible through the corporate firewall or a STUN/TURN server will have to be operated by the enterprise. In the latter case, it is not clear how the right STUN secret will be known to the JS run on the browser.

In any case, as these values will be widely known, we should regard them as obsfuscation and not as a serious security mechanism. That means that we cannot depend on authentication of STUN or TURN messages and need to watch for any assumptions that this is a strong form of authentication.

Since STUN and TURN have not yet seen the level of widespread use expected to be caused by WebRTC, the IETF has very recently (Feb 2014) approved the formation of a new working group (TRAM - TURN Revised and Modernized[2]) to investigate any changes that may be needed to STUN and TURN for the WebRTC environment. This work may improve on the security situation described here but has yet to get started at the time of writing.

4.4 DTLS-SRTP Keying

DTLS-SRTP sessions will be setup between the browsers as the MTI option for media security. These sessions will protect the media flows and DataChannel traffic. The SDP for the session is used to establish which browser plays server and which plays client in TLS terms.

Details of the specific ciphersuites that will be required and how those will map to one or more DTLS sessions and how those in turn will map to one or more sets of DTLS and SRTP keys are still emerging. One can imagine that the IETF WGs involved will want to ensure (for performance reasons) that only one DTLS handshake will be required so that the expense can be amortised over the various audio, video and data channels in both directions.

There is an almost implicit assumption that there will be IdP assertions related to the certificates used for TLS session however, details of the key generation and registration (in PKI terms) involved are not yet set down.

Gateways to legacy services and for transcoding will likely use long term DTLS server private keys and server authenticated sessions but for the main browser-to-browser use cases the situation is less clear.

It is also not yet clear if browsers will have an option to use ephemeral keys or not, nor whether some form of key continuity will be achieveable given the portability requirements that WebRTC involves. This is a standard PKI private key portability issue and has known consequences, though is not an easy problem to solve. Expecting WebRTC IdPs to be able to manage the PKI data structures may turn out not to be viable, in which case such key continuity mechanisms may become attractive.

WebRTC inherently involves multiple ports and quite different protocol stacks namely HTTP and JS, the DTLS handshake but also SRTP packet processing or SCTP/DTLS encapsulated DataChannels. If there are no restrictions requiring all connections to use the same UDP port then there may well be interesting implementation challenges in securely sharing keying material between different processes listening on those ports, in particular for real servers (as opposed to browsers acting in the TLS server role).

[2] `http://tools.ietf.org/wg/tram/`

Chapter 5

Pervasive and Other Monitoring

WebRTC has the potential to affect the extent and ubiquity of monitoring, either in a positive or negative way, regardless of one's point of view.

If you consider Legal Interception (LI) a good thing, then you might be concerned about WebRTC as it de-centralises the signalling required for interpersonal communications and has built in cryptography (via DTLS) that is specified to be turned on by default, albeit with ephemeral keying which may be vulnerable to MITM attacks. Similarly, if one considers Pervasive Monitoring (PM) as a "super-LI" capability, WebRTC encryption places barriers in the way of implementing PM (of media) via simple passive monitoring.

If, on the other hand, you are more concerned with individuals having freedom from pervasive monitoring then you might be concerned about WebRTC as it will mean that browsers by default will include the capability to listen and watch users, and a permissions model that is very likely to not be understood by many users. In addition, WebRTC media, though protected via DTLS will often not be sent between two browsers only but between a browser and one or more servers, at which servers all the usual LI taps can be implemented. One might consider that given the ease with which a random web site can be hacked, that many more conversations can be recorded, either by Law Enforcement Agencies (LEAs) or by private, possibly criminal, actors.

The use of ICE in WebRTC might also provide a point at which traffic meta-data can be captured, which is often of more interest to LEAs and Signals Intelligence agencies.

Similarly, WebRTC IdP's are in an excellent position to capture and store meta-data about WebRTC calls. Given that the expectation is that most users will use large IdPs, these services will be in an excellent position to gather large amounts of call meta-data, which will then be vulnerable to commercial privacy-invasive exploitation. Such large data sets are also attractive for LEAs and Signals Intelligence agencies, who may hack or coerce IdPs or with whom IdPs may collude.

Given that web sites control WebRTC call setup via Javascript (JS), commercial entities operating web sites can also use WebRTC as a way of enhancing their privacy invasive collection of information about users, for example, by recording call meta-data and reporting this back to the web site where it can be centralised and/or sold to advertising agencies. Given the Same Origin Policy (SOP), the JS required certainly has the capability to do this, and given the number of web sites on the Internet, this is certain to be attempted and very hard to protect against. A normal user certainly has no chance to detect such actions and web sites are inherently less constrained in doing such, when compared to regulated telcos.

Finally, the screen sharing capabilities of WebRTC, if enabled by a user, could enable a site to watch everything done by the users/browser for extended periods, e.g. via taking snapshots of the screen as the user carries out actions such as banking that the user would consider sensitive. To mitigate this, at least one major browser vendor is currently only planning to enable WebRTC screen sharing via an installed browser extension, however it is not clear whether or not such restrictions will be effective - once one site is sufficiently attractive for a user to install such an

extension, then any other site can try use it too.

In this section we delve into these issues in more detail and plan to engage with the IETF and W3C working groups on the topics arising.

5.1 WebRTC Making LI Harder - DTLS

The use of DTLS for WebRTC means that middleboxes will not be passively able to capture media traffic. While a MITM attack can be mounted in such cases, the likelihood of discovery is probably unacceptably high, in particular for LI and perhaps even for PM. Mechanisms could be developed that would allow an "observatory" similar to the SSL observatory[1] to detect some active MITM attacks.

The use of DTLS should also make PM harder, at least when it comes to media capture. However, use of Variable Bit Rate (VBR) codecs is known [58] to enable recovery of voice traffic even with strong encryption and there is the potential for similar attacks with mixed media-streams, for example, one would assume that signals intelligence agencies would devote effort to being able to recover call information from mixed audio and video even when those are encrypted via DTLS. We know of no practical results here at present though.

5.2 WebRTC Making LI Harder - Distributed Signalling

The inherently distributed signalling of WebRTC means that it will be a challenge for an LEA to produce a "warrant" or other legal instrument requiring either media or signalling ("pen trace") related to calls to/from a given user agent or IP address or user to be collected.

While distributed signalling should also make PM harder, that may depend on the deployment choices made by web sites to some extent. If for example, many web sites make use of the same JS code, loaded from a small number of sites, that creates an attractive point of attack for PM, if one could infect the JS libraries with malware intended to enable PM. While DTLS keying will hopefully not be so vulnerable in this respect (as the implementation will be part of the browser), if SDES features remain as an option that can be chosen by JS code, then it could be possible to selectively feed the wrong JS to browsers (via attacks such as QUANTUMINSERT [12]) thus enabling export of some media keys.

5.3 How LI Might be done in WebRTC

When WebRTC involves a server-side node taking part in media exchanges (e.g., for a customer support call) then "normal" LI and call recording is trivial, though it may require the LEA to interact with the web site.

However, if many web sites use 3rd party providers for such calls, then the situation for LI may be quite similar to today's for such calls.

For browser-to-browser calls, if the web site has the option to use SDES, then LEAs are likely to insist that this be used in some cases. How this will evolve in different jurisdictions remains to be seen. It is thus critical that the relevant standards are clear about the capabilities of sites to use SDES, and may even be justified to consider if browsers ought be able to indicate which kind of keying is being used. However, such a debate is likely to be difficult, especially within the IETF, given the IETF's policy on wiretap [42]. That policy does come into play here as SDES use by web sites could also make calls vulnerable to non LEA tapping.

[1] `https://www.eff.org/observatory`

5.4 How PM Might be done in a WebRTC World

PM by state actors should in principle be made more difficult by WebRTC, given that call signalling is devolved to web sites rather than being under the control of telcos with a history of collaborating with LEAs and signals intelligence agencies. The default use of DTLS-SRTP should also mitigate PM, at least as far as media content is concerned. However, as we have seen signals intelligence agencies are willing to devote significant effort to PM and will presumably seek to re-establish tools that enable PM as and when they can.

For meta-data capture, the use of DTLS-SRTP is effectively meaningless however, and connections between browsers can be logged in the normal way.

So absent some lower layer encryption that mitigates PM, WebRTC will remain as vulnerable to meta-data capture as prior technologies.

5.5 How Permissions will allow a Confused User to be Monitored

Monitoring of users by commercial entities will however be made easier by WebRTC. Many of those entities will be monitoring for commercial reasons, for example, to build (and possibly sell) user profiles.

The permissions model for WebRTC is vulnerable in that permission revocation will probably require either zero'ing all browser data or else re-visiting the site, neither of which are at all likely to be common.

This means that sites can tempt users into granting permissions and then continue to make use of those for tracking purposes essentially indefinitely.

Technical mitigations for this problem are difficult since it is hard to envisage a UI that both allows users to easily say yes when they want to communicate, but also makes it easy to revoke permissions when those are no longer needed.

5.6 ICE as a Meta-Data Capture Tool

The use of STUN/TURN for NAT traversal adds a new server at which monitoring of meta-data becomes possible, and perhaps even easy.

Given that many web sites will not want to operate their own STUN/TURN infrastructure it seems likely that ISPs, IdPs and/or other large commercial entities will operate these parts of the WebRTC infrastructure thus centralising much call meta-data.

5.7 IdPs as a Meta-Data Capture Service

Larger IdPs will be in an excellent position to monitor WebRTC usage by users. There is again a real technical tension here though – smaller web sites and IdPs have been proven over and over to not be so good at maintaining the security of user data, nor of continued operation. This leads to larger IdPs, and again that centralisation trend works to assist monitoring.

5.8 JS as a Pervasive Monitoring Tool

Most sites will neither develop nor host the set of JS libraries that are used for WebRTC applications and will simply configure "standard" libraries and add their own look and feel. This means that those libraries will be constantly downloaded (and updated) from yet another small set of large operators. Inclusion of tracking code within these libraries is to be expected and is an attractive target for commercial monitoring.

Chapter 6

Application-layer (XSS) Attacks

In this Section we take a look at the impact of a Cross-Site Scripting (XSS) vulnerability within a WebRTC application on the security of a WebRTC call. In order to do so, we revisit the main building blocks of such a WebRTC application, the underlying XSS attacker model and the different capabilities an attacker gains by executing a successful attack.

6.1 Application scenario

When initiating a WebRTC call a Web application conducts the following steps (See Figure 6.1 for details):

1. **Request access to the user's media:** In order to gain access to the user's media a Web application may utilize the browser-provided *getUserMedia* API, which returns a stream of the requested media (audio or video or both). As soon as the method is invoked by a piece of JavaScript the user is asked for permissions to share her Webcam and microphone with the Web application.
2. **Initiating the peer-to-peer connection** After receiving the so called *MediaStream* the Web application has to attach the stream to an *RTCPeerConnection* object that takes care of the peer-to-peer connection between the actual browser and the remote browser.
3. **Providing a signaling channel:** In order to enable the *RTCPeerConnection* to establish the connection, the Web application is in need of providing an out-of-band signaling channel with the remote browser. Via the signalling channel the *RTCPeerConnection* is able to exchange all the necessary details needed to establish a direct peer-to-peer connection between the caller and the callee.
4. **Process remote stream** After the connection has been established, the *RTCPeerConnection* provides the remote *MediaStream* to the Web application. It can be used to process the transmitted content in any way (e.g. attach the stream to a canvas object to render video and play sound).

Important to note here, is the fact that permissions of the user are only necessary for the first step of accessing the user's camera or microphone. Permissions are not necessary for any other step.

6.2 Attacker model

We assume that some WebRTC-based Web applications will sooner or later suffer from traditional Web vulnerabilities such as XSS. Hence, it is important to highlight the impact of such a

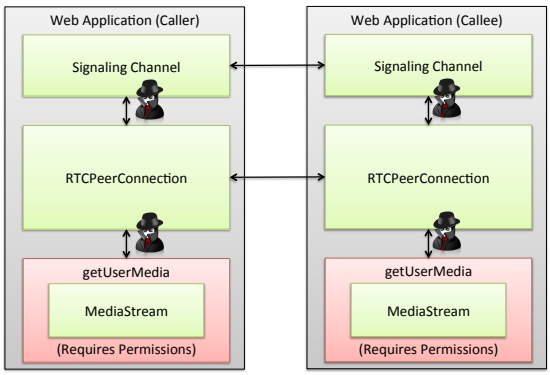

Figure 6.1: WebRTC overview

vulnerability on the overall security of WebRTC.

XSS is an attack in which an attacker is able to inject arbitrary JavaScript code into an existing Web application. The attacker's JavaScript therefore executes in the same security origin as the legitimate scripts of the application. As a consequence, the malicious JavaScript is able to access, overwrite, delete or create arbitrary DOM or JavaScript properties. In each step of the application scenario, the legitimate application has to call/access one or more JavaScript APIs or DOM elements. The APIs then invoke native functionality written in C/C++ within the browser that takes care of the technical protocols (e.g., for connecting to another host, playing sound, etc.). While the malicious JavaScript is not able to tamper with the native code itself, it can override the corresponding DOM APIs and prevent a call to native code in the first place. Afterwards the malicious code may invoke the original APIs with different parameters or in different order.

So as described above, in the following scenarios we consider an attacker that is capable of injecting arbitrary JavaScript code into an existing Web application:

1. **Third-party XSS attacker:** A third-party XSS attacker is capable of injection of a malicious payload through an existing XSS vulnerability. In order to conduct an attack, the adversary creates the payload and lures the victim onto an attacker controlled Web site that launches the attack.

2. **Malicious WebRTC application:** A malicious WebRTC application offers its users a WebRTC service, but aims at eavesdropping on its user. As the application itself provides the underlying "legitimate" functionality it is also able to add potentially malicious code without the user's consent.

3. **Malicious user/browser:** Even if the Web application is not malicious and there is no XSS vulnerability within the Web application, it is still possible to manipulate the JavaScript running within the Web site. So, for example, a user who wishes to record a call without the consent of the remote peer, could use standard JavaScript consoles to add a recording script to the legitimate calling site. Also browser extensions could inject scripts into a Web page to either record or eavesdrop on a WebRTC call.

6.3 Attack Scenarios

From the attacker model described above, we can deduce several points in time were an attacker's payload is able to alter the described process:

1. When the payload executes
2. When the user grants permissions to the legitimate application
3. When the signaling channel is used
4. When the connection is established

In the following we will describe four attack scenarios that we've implemented and that can be conducted by an attacker at one of the mentioned points in time. (Note: Hereby, we will not focus on traditional attack scenarios, such as cookie stealing, session hijacking, but we will focus on WebRTC-related attacks only).

6.3.1 Abuse of permissions

As soon as the attacker is able to lure a victim into clicking on a prepared link, his malicious payload executes in the context of the WebRTC application. As it is able to access any DOM API, the attacker is able to request permissions for the user's camera and microphone. Hence, the user is prompted to provide the permission to the *legitimate* application. If the user trusts the application and grants access to her camera, the malicious code receives a stream object that it is able to process in an arbitrary way.

As described in Chapter 3 permissions in Chrome are persistent as soon as the application utilizes HTTPS. So if an HTTPS-based application is vulnerable to XSS, the attacker's code immediately receives access to the MediaStream if the user granted these permissions before.

Similarly, the attacker's code could simply "wait" for the application to request permissions within a legitimate use case and receive the *MediaStream* afterwards without raising the user's suspicion. All of these attacks either take place shortly before or after step one of our application scenario.

6.3.2 Man-in-the-Middle attacks

After initiating the *RTCPeerConnection* object, a WebRTC-based application establishes an out-of-band signalling channel with the remote browser in order to exchange the technical details that are necessary to establish the direct peer-to-peer connection. If an attacker, is able to successfully inject her malicious payload into the application, the attacker is able manipulate the code responsible for establishing the signalling channel. Hence, a XSS attacker is capable of manipulating the message exchange and is therefore able to provide fake information to the *RTCPeerConnection* object. As a consequence the attacker is able to control to which remote host the Web application will connect to and thus the adversary is able to conduct man-in-the-middle attacks.

6.3.3 Information leakage attacks (full streams)

Besides a full man-in-the-middle attack, a XSS attacker is also able to simply copy the *MediaStreams* of a legitimate call. As stated in the specification a *MediaStream* "can be cloned" by a piece of JavaScript [15]. Hence, when attacking a victim the malicious payload simply needs to wait until the victim established a call to a remote peer, to clone the underlying local and remote *MediaStreams*. As creating additional WebRTC connections does not require any permissions, the attacker is able to connect to an attacker controlled host and to transmit the cloned MediaStreams while the legitimate call is in progress. Without spotting and analyzing the XSS payload, the user is not able to notice that a second transmission is ongoing.

6.3.4 Information leakage attacks (images)

Another way of leaking information from a WebRTC call is to steal screenshots of an ongoing transmission. After establishing a call, a *MediaStream* is often attached to an HTML5 Canvas element for displaying video and sound. The canvas element offers a screenshot feature to the surrounding JavaScript environment. Hence, at any time a legitimate or malicious piece of JavaScript is able to capture arbitrary images.

Within our prototypical implementation our XSS attack code creates a screenshot every 500ms and transmits it to a remote host via XMLHttpRequests and Cross-Origin Resource Sharing (CORS).

Chapter 7

Potential Confidentiality Leaks

7.1 High-level view of the attacks

7.1.1 Applicable attacker model

In this chapter, we consider the *Web attacker* model:

Web Attacker: *A potential victim of an attack visits, using his Web browser, a unrelated Web page, that is under the control of the adversary. Now, the adversary is able to execute arbitrary JavaScript in the browser of the potential victim in the context of the origin of the attacker's page.*

Please note the difference from the *XSS attacker* that is the subject of Chapter 6: The XSS attacker is able to execute JavaScript in the context of a vulnerable site, that utilizes WebRTC. The *Web attacker* is confined to an origin, that is directly and legitimately under his control.

The capabilities of the Web attacker lie within the statefulness of the Web browser. A browser implicitly carries state in respect to, for example:

- Authentication context: Valid session cookies from existing authentication contexts, cached HTTP authentication credentials, or established TLS tunnels to external hosts, which potentially were established using TLS client authentication.
- Local network characteristics, mainly represented via the browser's IP, which might be shadowed via Network Address Translation.
- Personalization with preconfigured identity provider contexts, such as Mozilla Persona or related approaches.
- Network behaviour, due to the existence of configured, client-side Web proxies.

Any of these characteristics can potentially be abused by the Web attacker for privilege escalation [23] or privacy violations [54].

7.1.2 Attack pattern

In the context of WebRTC and this chapter we consider the following basic attack pattern:

1. The victim visits an attacker-controlled Web page. This visit could have happened by chance, due to social engineering by the attacker (e.g., via a deceitful email), or because the attacker was able to compromise the client-side code of a Web site, which the victim visits regularly. For the latter scenario, in many cases it suffices for the attacker to purchase advertising space on a popular commercial site [44].

2. As soon as the victim's browser renders the attacker's content, the included JavaScript initiates a WebRTC `RTCPeerConnection` to a communication partner under the control of the attacker.

3. In this way, the attacker can create a condition, in which he controls both the signalling infrastructure and both sides of the communication channel. In such a position, he is able to access meta information that is implicitly bound to the communication process.

Potential leakage of security sensitive information can happen both on the network layer (see Sec. 7.2) and on the application layer (see Sec. 7.3).

7.2 Local IP disclosure

7.2.1 High-level view

As discussed before, the majority of the WebRTC protocol specifics are realized via JavaScript APIs (as opposed to a low-level, native implementation in the browser). As a result, most of the meta data occurring is available either within the JavaScript space or at the signalling infrastructure.

Hence, as the attacker fully controls both the JavaScript that is executed in the context of the calling page, the utilized signalling servers, and the remote host that is contacted, he is able to access any information that occurs because of the communication attempt. Included in this set of available information is the browser's *local IP address*, which is of potentially high interest to the attacker. Please refer to Sections 7.2.4, 7.2.5, and 7.2.6 for discussions in respect to potentially malicious usage of the local IP address.

7.2.2 Specific attack: Local JavaScript

The WebRTC setup is done in the browser via JavaScript and its respective APIs (such as `RTCPeerConnection`). This design choice provides a high degree of flexibility and control to sites that want to use the technology legitimately. However, this means that a lot of low-level meta information present is exposed to the JavaScript space. As mentioned above, JavaScript can obtain the browser's local IP. Listing 1 exemplifies this approach of this technique, which is realized by first creating a fresh `RTCPeerConnection` object and then parsing the desired information out of the ICE candidate record.

Listing 1 Local IP leakage (simplified sketch)

```
var rtc = new RTCPeerConnection({iceServers:[]});
rtc.onicecandidate = function (evt) {
    if (evt.candidate){
      readLocalIP(evt.candidate.candidate)
    }
};

function readLocalIP(candidate){
  // Obtain the local IP from the ICE candidate record
  // [...]
}
```

Figure 7.1 shows an example Web page, that utilizes this functionality to practically obtain the user's local IP address. Note, that the browser window in in "incognito" mode, a specific mode that is supposed to hide all potentially private state from the displayed Web page.

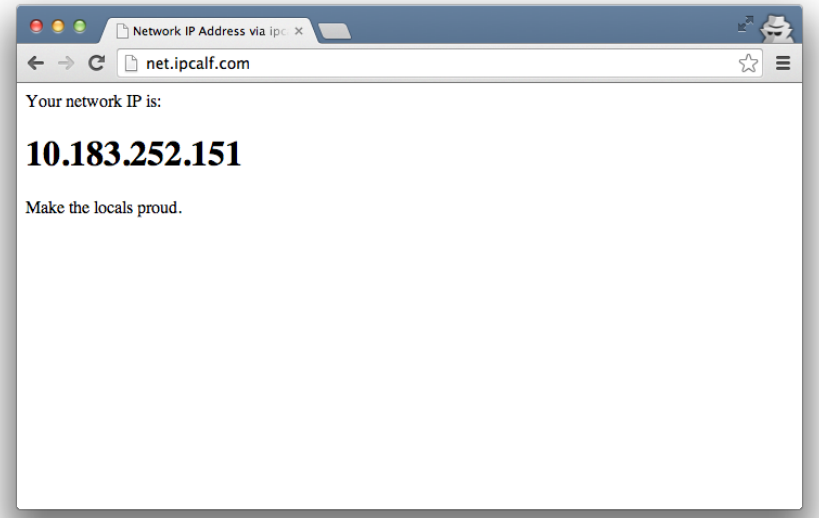

Figure 7.1: Leakage of local network information

7.2.3 Specific attack: Controlling the signalling infrastructure

Furthermore, as mentioned above, the attacker is able cause the victim's browser to use signalling infrastructure that is under the control of the adversary.

This especially includes any STUN and TURN servers used. In Listing 1, this would be done in the first line of the code, in the `iceServers:[]` parameter.

As a consequence, the meta data present is available to the attacker not only via JavaScript but also as part of the connection setup network traffic, which traverses through the attacker's servers.

7.2.4 Security implications: Intranet attacks

Firewalls are an indispensable staple in today's network architecture. They provide strong protection of internal hosts and systems against remote attackers. For most purposes, a firewall can be regarded as tool for access control: Only communication partners, that are within the local network (and, hence, are implicitly trustworthy) are granted access to the internal systems. All parties outside of the firewall's boundaries are automatically denied access. Figure 7.2 shows this simple, yet effective protection approach.

However, every end-user computer in our networks runs one program, that receives computer code from the outside and runs this code within the boundaries of the local net: The Web browser, that is executing JavaScript from unvalidated and potentially malicious sources. Over the years, JavaScript's capabilities steadily grew, especially in respect to network access. Thus, if used correctly, an attacker could attempt to compromise the local network from within, using a user's browser (see Figure 7.3). Over the years, several attacks in this area have been disclosed and discussed, ranging from port-scanning the intranet to fully compromising unpatched servers [48].

A main obstacle in successfully executing such attacks is the missing knowledge on the intranet's network specifics, foremost amongst this information is the applicable IP range. Up to now, this information was well hidden from the active code within the browser, as JavaScript does not possess low-level networking abilities on the IP layer. Hence, the only IP the attack sees, is the external IP of the network's firewall, while the actual IP is hidden behind Network

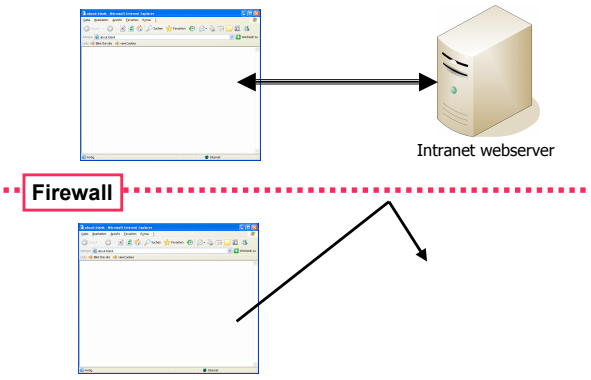

Figure 7.2: Using firewalls to protect internal hosts

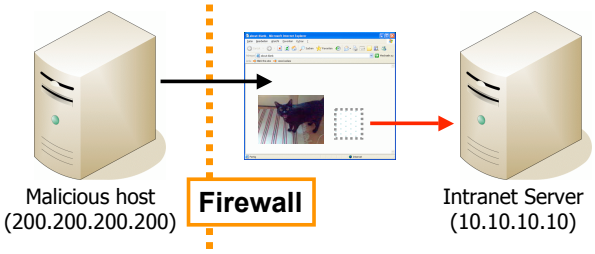

Figure 7.3: Firewall circumvention via the Web browser

Address Translation (NAT).

However, with WebRTC, this is no longer the case. As discussed in Sections 7.2.2 and 7.2.3, WebRTC reliably allows the attacker to gain information about the browser's local IP. In Figure 7.1, an example is shown in which the JavaScript was able to obtain the information that the browser's IP is within the private RFC 1918 10.*.*.* range.

7.2.5 Security implications: De-anonymization

Privacy conscious users frequently use privacy enhancing anonymization tools, such as TOR. TOR routes the network traffic through a set of Web proxies, effectively hiding the actual network information from the accessed servers. On the client-side, TOR works by being configured as a local proxy for the Web browser.

Unfortunately, the WebRTC-based attacks discussed here weaken the privacy guarantees of anonymization tools. To create a browser-to-browser channel, the actual network specifics need to be utilized and communicated. Hence, they are available to any Web site rendered by the browser, regardless of the TOR exit node currently used.

Especially in long-lived local network setups, in which the user's local IP rarely changes or in situations, in which only a subset of all sites are accessed via anonymized channels, the methods outlined can be used by the attacker to capture observed network traffic containing the local IP address of browsers, thus, effectively allowing de-anonymization.

7.2.6 Security implications: User tracking

Closely related to the de-anonymization attack, are all utilizations of the local network information to track internet users. The local IP address is a characteristic that is not under the control of the browser and cannot be masked or suppressed. Hence, even if privacy supporting

browser add-ons are used, sites that try to track the user for profiling purposed can use this characteristic as a strong indicator in the user's fingerprint [54].

7.2.7 Potential remedies

At this point, there is no straight-forward solution to the identified problem. The local IP address is essential information for the signalling infrastructure (which is under the control of the attacker). Hence, even if the protocol implementation within the JavaScript space could be altered to hide certain information from the script code, it would most likely still be available to the attacker.

Thus, probably the most promising option would be to introduce a new class of permissions (see Chapter 3) that would require explicit user consent whenever a page tries to initiate a RTCPeerConnection.

7.3 Disclosure of precise user identity information

7.3.1 Basics of the WebRTC identity provider mechanism

The WebRTC architecture is highly flexible. A notable characteristic of WebRTC is the decoupling of the identity providers and parties involved in setting up the connection. In practice, this means that site `poker.example.com` sets up a WebRTC connection for a game between Alice, who used `alice@twitter.com` as her identity and Bob, who uses `bob@facebook.com` as his identity.

From this setting, it is immediately obvious, that none of the three parties that are directly involved has the capabilities to verify the identities of the communication partners involved in the browser-to-browser channel. However, there is a need for a mechanism that allows Alice to verify that she is indeed talking to Bob. For this purpose, the WebRTC security architecture [63] defines an open identity provider mechanism.

High-level overview: To verify that she is indeed talking to `bob@facebook.com`, Alice can trigger an *identity assessment*. This process binds the open WebRTC *PeerConnection* to an identity assertion that is verifiably signed by `facebook.com`. In case Alice requires such a proof of identity, Bob's browser executes a dedicated JavaScript from `facebook.com` in an isolated execution environment. This JavaScript verifies the credentials that Bob shares with his identity provider, and creates an assertion record that binds the existing *PeerConnection* to Bob's identity. After receiving this assertion, Alice can verify, that she indeed is connected to Bob, (according to facebook.com) without interference from the signalling infrastructure setup of `poker.example.com`.

7.3.2 Potential abuse of the mechanism

As motivated in Section 7.1, a potentially malicious site can setup WebRTC connections without the consent and knowledge of the attacked user. In such a setting, the adversary controls all aspects of the connection setup. Hence, he is able to trigger identity assertion creation in the victim's browser.

This can cause a privacy problem in scenarios in which the victim's default identity provider is configured statically in the browser (as it is the case with, for example, the former Mozilla Persona proposal). See the W3C draft [10] on this topic:

> For purposes of generating assertions, the IdP shall be chosen as follows:
>
> 1. If the setIdentityProvider() method has been called, the IdP provided shall be used.

2. If the setIdentityProvider() method has not been called, then the browser shall use an IdP configured into the browser. If more than one such IdP is configured, the browser should provide the user with a chooser interface.

Thus, in a situation in which a user has exactly one identity provider configured in his browser (a fairly likely scenario until the corresponding technologies become more established), an attacker can acquire identity assertions without any prior knowledge of the victim's identity. After the assertion is complete, the corresponding JavaScript API allows convenient access to the victim's identity (see Listing 2).

Listing 2 JavaScript access to the victim's identiy

```
pc.onidentityresult = function(result) {
  console.log("IdP= " + pc.peerIdentity.idp +
              " identity=" + pc.peerIdentity.name);
};
```

Hence, in the current situation, every Web site, regardless of it's effective trust relationship to the user, can obtain the user's identity in the case that he has an identity provider bound to his Web browser.

7.3.3 Potential remedies

In a similar way to the security problem discussed in Section 7.2, the most promising mitigation for this problem would be obtaining explicit user consent before accessing his identity information. Through a user dialogue, such as *"The site* `poker.example.com` *would like to access your identity information (allow/deny)"*, a user could communicate his consent to the Web site's identity assertion. Any attempt by an untrustworthy Web site to access the information in an unexpected setting would be immediately apparent to the user and he would have suitable means to stop the information leak.

Chapter 8

Summary

We have seen how the WebRTC architecture represents an evolution of the Web architecture and of VoIP handling. But this is an evolution that could have quite far-reaching impact if the predicted levels of deployment of WebRTC are reached.

We have started the process of testing the methodology from STREWS D1.1 in this case study. That work will progress further as the project develops and as WebRTC standardisation progresses beyond recent stagnant discussions of video codec intellectual property.

The WebRTC security architecture is complex, and complexity being the enemy of security, we expect that the path towards a robustly secure and widely-deployed WebRTC will likely be less smooth than currently envisaged.

The WebRTC architecture continues the centralising meta-theme seen in many other areas of Internet endeavour - basically where larger and larger scale services are favoured by the architecture, to the point where some dominant players inevitably emerge. With WebRTC in addition to the few browser implementations carrying a lot of influence, there may be few enough core JS libraries that get wide deployment, only a handful of IdPs and even fewer Internet-scale TURN services. On, the other hand, WebRTC does also hold out the promise that some form of target dispersion could work given the ubiquity of support expected in browsers and Web servers. But it seems highly likely that such deployments will only form the long-tail that marches off to the right of the few services that have some control over the overwhelming majority of WebRTC traffic.

We have highlighted some specific aspects of WebRTC that we feel are worthy of further study, within and beyond STREWS. At a high level these are

- how permissions are to be handled,
- how WebRTC will interact with pervasive monitoring,
- key management for WebRTC, and
- the impact of cross-site vulnerabilities, which despite being widely known are still common on all web sites and will be common on sites offering WebRTC services.

Aside from these considerations, we have also identified some specific vulnerabilities which will be brought to the attention of relevant IETF and W3C working group participants. While mitigating these vulnerabilities may not be entirely within the scope of those groups (which are not responsible for specific implementations), we expect that considering these vulnerabilities will lead to improvements in the output of the standards groups.

WebRTC security does however represent an interesting attempt to move beyond the current IETF process limitation, whereby MTI security features are mandated but usage is optional, to a situation where strong security features must not only be implemented but offered as a default choice. That facet may in the end be the most important one - widely deployed but weak or

broken security mechanisms get fixed and improve relatively quickly. The security improvements of WPA over WEP in the context of wi-fi are probably the canonical reference for this argument. In contrast, non-implemented or unusable security mechanisms tend to be ignored until there is a serious crisis. TLS client authentication on the public Web is perhaps a good example here - even though there are some niche applications for this, the deployability of the mechanism has not really progressing in nearly two decades.

However, that said the Web and security communities have learned significant lessons about usability and scaling in those decades, so one can be optimistic that the applications of those lessons will result is more speedy progress towards a broadly deployed but robustly secured WebRTC.

References

Bibliography

[1] Convention for the protection of human rights and fundamental freedoms as amended by protocols no. 11 and no. 14. 1950.

[2] Investigation inot the processing of personal data for the 'whatsapp' mobile application by whatsapp inc. jan 2013.

[3] H. Alvestrand. Overview: Real Time Protocols for Browser-based Applications. Internet-Draft draft-ietf-rtcweb-overview-11, Internet Engineering Task Force, August 2014. Work in progress.

[4] H. Alvestrand. Transports for WebRTC. Internet-Draft draft-ietf-rtcweb-transports-06, Internet Engineering Task Force, August 2014. Work in progress.

[5] Laura Arribas, Frederick Hirsch, and Dominique Hazaël-Massieux. Device API Access Control Use Cases and Requirements. *W3C Working Group Note*, March 2011.

[6] J. Bankoski, J. Koleszar, L. Quillio, J. Salonen, P. Wilkins, and Y. Xu. VP8 Data Format and Decoding Guide. RFC 6386 (Informational), November 2011.

[7] Jim Barnett and Travis Leithead. MediaStream Recording. *W3C Working Draft*, February 2013.

[8] M. Baugher, D. McGrew, M. Naslund, E. Carrara, and K. Norrman. The Secure Real-time Transport Protocol (SRTP). RFC 3711 (Proposed Standard), March 2004. Updated by RFCs 5506, 6904.

[9] M. Belshe, R. Peon, and M. Thomson. Hypertext Transfer Protocol version 2. Internet-Draft draft-ietf-httpbis-http2-14, Internet Engineering Task Force, July 2014. Work in progress.

[10] Adam Bergkvist, Daniel C. Burnett, Cullen Jennings, and Anant Narayanan. WebRTC 1.0: Real-Time Communication Between Browsers. *W3C Working Draft*, 2012.

[11] Robin Berjon, Steve Faulkner, Travis Leithead, Erika Doyle Navara, Edward O'Connor, Silvia Pfeiffer, and Ian Hickson. HTML 5.1 Specification. *W3C Working Draft*, 2013.

[12] Didier Bigo, Sergio Carrera, Nicholas Hernanz, Julien Jeandesboz, Joanna Parkin, Francesco Ragazzi, and Amandine Scherrer. *Mass Surveillance of Personal Data by EU Member States and its Compatibility with EU Law. CEPS Liberty and Security in Europe No. 61, 6 November 2013.* 2013.

[13] Greg Billock, James Hawkins, and Paul Kinlan. Web Intents. *W3C Working Group Note*, May 2013.

[14] Ian Brown. Wittness statement of dr. ian brown, sep 2013.

[15] Daniel C. Burnett, Narayanan Anant, Adam Bergkvist, and Cullen Jennings. Media Capture and Streams. *W3C Working Draft*, September 2013.

[16] Daniel C. Burnett, Narayanan Anant, Adam Bergkvist, and Cullen Jennings. Media Capture and Streams. *W3C Editor's Draft*, August 2014.

[17] G. Camarillo, O. Novo, and S. Perreault. Traversal Using Relays around NAT (TURN) Extension for IPv6. RFC 6156 (Proposed Standard), April 2011.

[18] G. Camarillo and H. Schulzrinne. Early Media and Ringing Tone Generation in the Session Initiation Protocol (SIP). RFC 3960 (Informational), December 2004.

[19] R. Clayton and M. Kucherawy. Source Ports in Abuse Reporting Format (ARF) Reports. RFC 6692 (Proposed Standard), July 2012.

[20] Alissa Cooper, Frederick Hirsch, and John Morris. Device API Privacy Requirements. *W3C Working Group Note*, June 2010.

[21] D. Crocker. Internet Mail Architecture. RFC 5598 (Informational), July 2009.

[22] D. Crocker and P. Resnick. STRINT Workshop Position Paper: Levels of Opportunistic Privacy Protection for Messaging-Oriented Architectures. Internet-Draft draft-crocker-strint-workshop-messaging-00, Internet Engineering Task Force, January 2014. Work in progress.

[23] Philippe De Ryck, Lieven Desmet, Thomas Heyman, Frank Piessens, and Wouter Joosen. CsFire: Transparent client-side mitigation of malicious cross-domain requests. In *Engineering Secure Software and Systems*, pages 18–34. Springer, 2010.

[24] Lieven Desmet and Martin Johns. Real-time communications security on the web. *IEEE Internet Computing*, 18(6), 2014.

[25] Lieven Desmet and Frank Piessens. Web-platform security guide. *Strews Deliverables*, (D.1.1), November 2013.

[26] T. Dierks and C. Allen. The TLS Protocol Version 1.0. RFC 2246 (Proposed Standard), January 1999. Obsoleted by RFC 4346, updated by RFCs 3546, 5746, 6176.

[27] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), August 2008. Updated by RFCs 5746, 5878, 6176.

[28] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. Technical report, DTIC Document, 2004.

[29] C. Falls. Reservations for Network Group meeting. RFC 245, October 1971.

[30] S. Farrell, P. Hoffman, and M. Thomas. HTTP Origin-Bound Authentication (HOBA). Internet-Draft draft-ietf-httpauth-hoba-04, Internet Engineering Task Force, August 2014. Work in progress.

[31] S. Farrell and H. Tschofenig. Pervasive Monitoring Is an Attack. RFC 7258 (Best Current Practice), May 2014.

[32] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Obsoleted by RFCs 7230, 7231, 7232, 7233, 7234, 7235, updated by RFCs 2817, 5785, 6266, 6585.

[33] M. Handley and V. Jacobson. SDP: Session Description Protocol. RFC 2327 (Proposed Standard), April 1998. Obsoleted by RFC 4566, updated by RFC 3266.

[34] M. Handley, V. Jacobson, and C. Perkins. SDP: Session Description Protocol. RFC 4566 (Proposed Standard), July 2006.

[35] Dominique Hazaël-Massieux. Permissions for Web APIs. Jan 2014.

[36] Dominique Hazaël-Massieux. Standards for Web Applications on Mobile: current state and roadmap. January 2014.

[37] Frederick Hirsch. Web Application Privacy Best Practices. *W3C Working Group Note*, July 2012.

[38] P. Hoffman. The gopher URI Scheme. RFC 4266 (Proposed Standard), November 2005.

[39] P. Hoffman and J. Schlyter. The DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol: TLSA. RFC 6698 (Proposed Standard), August 2012. Updated by RFC 7218.

[40] C. Holmberg, S. Hakansson, and G. Eriksson. Web Real-Time Communication Use-cases and Requirements, February 2014. Expires: August 10, 2014.

[41] C. Holmberg, S. Hakansson, and G. Eriksson. Web Real-Time Communication Use-cases and Requirements. Internet-Draft draft-ietf-rtcweb-use-cases-and-requirements-14, Internet Engineering Task Force, February 2014. Work in progress.

[42] IAB and IESG. IETF Policy on Wiretapping. RFC 2804 (Informational), May 2000.

[43] Ecma International. ECMAScript Language Specification. March 2011.

[44] Collin Jackson, Adam Barth, Andrew Bortz, Weidong Shao, and Dan Boneh. Protecting browsers from dns rebinding attacks. *ACM Transactions on the Web (TWEB)*, 3(1):2, 2009.

[45] Markus Jakobsson. *The Death of the Internet*. Wiley, 2012.

[46] R. Jesup, S. Loreto, and M. Tuexen. WebRTC Data Channel Establishment Protocol. Internet-Draft draft-ietf-rtcweb-data-protocol-07, Internet Engineering Task Force, July 2014. Work in progress.

[47] R. Jesup, S. Loreto, and M. Tuexen. WebRTC Data Channels. Internet-Draft draft-ietf-rtcweb-data-channel-11, Internet Engineering Task Force, July 2014. Work in progress.

[48] Martin Johns. On JavaScript Malware and Related Threats - Web Page Based Attacks Revisited. *Journal in Computer Virology, Springer Paris*, 4(3):161–178, August 2008.

[49] A. Johnston, S. Donovan, R. Sparks, C. Cunningham, and K. Summers. Session Initiation Protocol (SIP) Basic Call Flow Examples. RFC 3665 (Best Current Practice), December 2003.

[50] Travis Leithead. MediaStream Capture Scenarios. *W3C Working Draft*, March 2012.

[51] R. Mahy, P. Matthews, and J. Rosenberg. Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN). RFC 5766 (Proposed Standard), April 2010.

[52] Giridhar Mandyam. Mediastream Image Capture. *W3C Working Draft*, July 2013.

[53] D. McGrew and E. Rescorla. Datagram Transport Layer Security (DTLS) Extension to Establish Keys for the Secure Real-time Transport Protocol (SRTP). RFC 5764 (Proposed Standard), May 2010.

[54] Nick Nikiforakis, Alexandros Kapravelos, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. Cookieless monster: Exploring the ecosystem of web-based device fingerprinting. In *IEEE Security and Privacy*, 2013.

[55] M. Nottingham and M. Thomson. Opportunistic Encryption for HTTP URIs. Internet-Draft draft-ietf-httpbis-http2-encryption-00, Internet Engineering Task Force, June 2014. Work in progress.

[56] EU Network of Independent Experts on Fundamental Rights. Commentary of the charter of fundamental rights of the european union. 2006.

[57] The European Parliament, The European Council, and The European Commission. Charter of Fundamental Rights of the European Union, March 2010.

[58] C. Perkins and JM. Valin. Guidelines for the Use of Variable Bit Rate Audio with Secure RTP. RFC 6562 (Proposed Standard), March 2012.

[59] C. Perkins, M. Westerlund, and J. Ott. Web Real-Time Communication (WebRTC): Media Transport and Use of RTP. Internet-Draft draft-ietf-rtcweb-rtp-usage-17, Internet Engineering Task Force, August 2014. Work in progress.

[60] M. Perumal, D. Wing, R. R, T. Reddy, and M. Thomson. STUN Usage for Consent Freshness. Internet-Draft draft-ietf-rtcweb-stun-consent-freshness-06, Internet Engineering Task Force, August 2014. Work in progress.

[61] Dave Raggett, Arnaud Le Hors, and Ian Jacobs. HTML 4.01 Specification. *W3C Recommendation*, 1999.

[62] E. Rescorla. Security Considerations for WebRTC. Internet-Draft draft-ietf-rtcweb-security-07, Internet Engineering Task Force, July 2014. Work in progress.

[63] E. Rescorla. WebRTC Security Architecture. Internet-Draft draft-ietf-rtcweb-security-arch-10, Internet Engineering Task Force, July 2014. Work in progress.

[64] E. Rescorla and N. Modadugu. Datagram Transport Layer Security Version 1.2. RFC 6347 (Proposed Standard), January 2012.

[65] J. Rosenberg. Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols. RFC 5245 (Proposed Standard), April 2010. Updated by RFC 6336.

[66] J. Rosenberg, R. Mahy, P. Matthews, and D. Wing. Session Traversal Utilities for NAT (STUN). RFC 5389 (Proposed Standard), October 2008.

[67] J. Rosenberg and H. Schulzrinne. An Offer/Answer Model with Session Description Protocol (SDP). RFC 3264 (Proposed Standard), June 2002. Updated by RFC 6157.

[68] P. Saint-Andre. Extensible Messaging and Presence Protocol (XMPP): Address Format. RFC 6122 (Proposed Standard), March 2011.

[69] P. Saint-Andre. Extensible Messaging and Presence Protocol (XMPP): Core. RFC 6120 (Proposed Standard), March 2011.

[70] P. Saint-Andre. Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence. RFC 6121 (Proposed Standard), March 2011.

[71] J. Schiller. Strong Security Requirements for Internet Engineering Task Force Standard Protocols. RFC 3365 (Best Current Practice), August 2002.

[72] H. Schulzrinne and S. Casner. RTP Profile for Audio and Video Conferences with Minimal Control. RFC 3551 (INTERNET STANDARD), July 2003. Updated by RFCs 5761, 7007.

[73] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications. RFC 3550 (INTERNET STANDARD), July 2003. Updated by RFCs 5506, 5761, 6051, 6222, 7022, 7160, 7164.

[74] Ashkan Soltani and Barton Gellman. Nsa infiltrates links to yahoo, google data centers worldwide, snowden documents say. *The Washington Post*, 2013.

[75] M. Tuexen, R. Stewart, R. Jesup, and S. Loreto. DTLS Encapsulation of SCTP Packets. Internet-Draft draft-ietf-tsvwg-sctp-dtls-encaps-05, Internet Engineering Task Force, July 2014. Work in progress.

[76] J. Uberti and C. Jennings. Javascript Session Establishment Protocol.

[77] J. Uberti, C. Jennings, and E. Rescorla. Javascript Session Establishment Protocol. Internet-Draft draft-ietf-rtcweb-jsep-07, Internet Engineering Task Force, July 2014. Work in progress.

[78] J. Valin and C. Bran. WebRTC Audio Codec and Processing Requirements. *IETF Internet Draft*.

[79] J. Valin and C. Bran. WebRTC Audio Codec and Processing Requirements. Internet-Draft draft-ietf-rtcweb-audio-05, Internet Engineering Task Force, February 2014. Work in progress.

[80] JM. Valin, K. Vos, and T. Terriberry. Definition of the Opus Audio Codec. RFC 6716 (Proposed Standard), September 2012.